2016-09

# Constructing and classifying email networks from raw forensic images

Allen, Gregory

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/50520

# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# THESIS

## CONSTRUCTING AND CLASSIFYING EMAIL NETWORKS FROM RAW FORENSIC IMAGES

by

Gregory Allen

September 2016

Thesis Advisor: Michael McCarrin
Thesis Co-Advisor: Ralucca Gera

**Approved for public release. Distribution is unlimited.**

THIS PAGE INTENTIONALLY LEFT BLANK

# REPORT DOCUMENTATION PAGE

*Form Approved OMB No. 0704–0188*

| | | |
|---|---|---|
| 1. AGENCY USE ONLY *(Leave Blank)* | 2. REPORT DATE<br>September 2016 | 3. REPORT TYPE AND DATES COVERED<br>Master's Thesis 09-01-2014 to 09-17-2016 |

4. TITLE AND SUBTITLE

CONSTRUCTING AND CLASSIFYING EMAIL NETWORKS FROM RAW FORENSIC IMAGES

5. FUNDING NUMBERS

6. AUTHOR(S)

Gregory Allen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release. Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Email addresses extracted from secondary storage devices are important to a forensic analyst when conducting an investigation. They can provide insight into the user's social network and help identify other potential persons of interest. However, a large portion of the email addresses from any given device are artifacts of installed software and are of no interest to the analyst. We propose a method for discovering relevant email addresses by creating graphs consisting of extracted email addresses along with their byte-offset location in storage. We compute certain global attributes of these graphs to construct feature vectors, which we use to classify graphs into "useful" and "not useful" categories. This process filters out the majority of uninteresting email addresses. We show that using the network topological measures on the dataset tested, Naïve Bayes and SVM were successful in identifying $100\%$ and $95.5\%$, respectively, of all graphs that contained useful email addresses both with areas under the curve above $.97$ and F1 scores at $.80$ and $.90$ for Naïve Bayes and SVM, respectively. Our results show that using network science metrics as attributes to classify graphs of email addresses based on the graph's topology could be an effective and efficient tool for automatically delivering evidence to an analyst.

14. SUBJECT TERMS

digital forensics, network science, graph theory, machine learning

15. NUMBER OF PAGES 101

16. PRICE CODE

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
| Unclassified | Unclassified | Unclassified | UU |

THIS PAGE INTENTIONALLY LEFT BLANK

# CONSTRUCTING AND CLASSIFYING EMAIL NETWORKS FROM RAW FORENSIC IMAGES

Gregory Allen
Lieutenant, United States Navy
B.S., University of Cincinnati, 2008

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
**September 2016**

Approved by: Michael McCarrin
Thesis Advisor

Ralucca Gera
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Email addresses extracted from secondary storage devices are important to a forensic analyst when conducting an investigation. They can provide insight into the user's social network and help identify other potential persons of interest. However, a large portion of the email addresses from any given device are artifacts of installed software and are of no interest to the analyst. We propose a method for discovering relevant email addresses by creating graphs consisting of extracted email addresses along with their byte-offset location in storage. We compute certain global attributes of these graphs to construct feature vectors, which we use to classify graphs into "useful" and "not useful" categories. This process filters out the majority of uninteresting email addresses. We show that using the network topological measures on the dataset tested, Naïve Bayes and SVM were successful in identifying 100% and 95.5%, respectively, of all graphs that contained useful email addresses both with areas under the curve above .97 and F1 scores at .80 and .90 for Naïve Bayes and SVM, respectively. Our results show that using network science metrics as attributes to classify graphs of email addresses based on the graph's topology could be an effective and efficient tool for automatically delivering evidence to an analyst.

THIS PAGE INTENTIONALLY LEFT BLANK

# Table of Contents

# List of Figures

# List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

# Acknowledgments

I would first like to express my sincere gratitude to my thesis advisors, Michael McCarrin and Dr. Ralucca Gera. Their support and guidance throughout the entire process was instrumental and never wavered. The amount of dedication and effort that they put into their research and learning of new ideas was truly inspiring.

I would also like to thank the faculty and students of my cohort at the Naval Postgraduate School. Their professionalism was un-matched and I would not have been successful without their support.

Finally to my daughter, Emma. You are my inspiration. May you never stop learning.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 1: Introduction

Forensic analysts must examine large collections of data to perform their work. Much of this entails sifting through the data to determine what information may be relevant and what information has no bearing on an investigation. Email addresses stored on a digital device are of particular importance when conducting an investigation. They provide information on who the user of the device is communicating with and can produce further leads. Unfortunately, many of the email addresses on a particular device are not informative—likely because they are artifacts of installed software, rather than indicators of communication between the user and members of his/her social network. Our research studies groups of email addresses that we represent as graphs based on their location in storage on a given device, and analyzes the attributes of those graphs to classify them by usefulness.

We use Network Science (see Section 2.1) heavily in our research. Networks are a group or system of objects that are interconnected by a relationship of some type [1], [2]. They show up everywhere. The Internet connects everything that is connected to the Internet. Facebook, Twitter, LinkedIn, and Meetup are all examples of social networks (defined below in section 2.1.1) in which users are able to correspond with other users of their choosing. The brain can be thought of as a network in which millions of neurons send signals to each other controlling functions of the body.

Networks range from small and simple, such as relationships in a family tree, where network behavior can be easily understood, to complex networks, where the number of elements are extremely large and the organizing principle may be difficult to identify. Network Science provides the framework for understanding networks in an attempt to extract emergent properties, understand the function of the system, be able to predict behavior of the network, and in some cases to be able to control how the network evolves.

Our hypothesis is that networks of email addresses stored on a device, constructed solely based on their proximity to each other in storage can be classified into being useful or not based on the network's topology. Throughout our paper, a *useful* network is defined as one containing email address that the user(s) of the device communicates with. We test our hypothesis against a dataset of 10 drives that were collected in previous research [3]. We look at the 20 largest connected components of email addresses from each storage device. We test over 40 attributes on each resultant graph and use machine learning to predict a graph's usefulness based on those attributes.

## 1.1 Motivation

The primary motivation of our research is to reduce the amount of time a forensic analyst takes in identifying useful email addresses on a device. A common approach is to obtain an image of a device and then use a forensic tool, such as EnCase or Autopsy to look through emails. This provides an analyst with easy access to data in a structure that is already familiar. The analyst can browse through contact lists, inboxes, folders of sent items, and others in order to create a social network affiliated with the user. Of course, the application may be password enabled, blocking unauthorized access, but with enough time, this restraint can often be mitigated. Often in an investigation however, time is a factor, and going through every possible email account a user might use takes time.

Much of this analysis can be automated. There are several digital forensics tools, such as FTK and the sleuth kit that extract information from a device and present it in a manner that is useful to the analyst. These tools are able to conduct powerful searches to help find relevant information. However, these tools are designed for specific operating systems or platforms and a given tool may not be effective on certain devices. Also, email addresses are sometimes saved in atypical locations where some digital forensic tools cannot recover [4]. We treat the task of filtering email addresses as a classification problem.

Our goal is to automate the work of accurately classifying whether or not a network of email addresses is useful and should be examined more closely, ultimately saving valuable time for the analyst. Our research focuses on answering the following:

- What is the best classification method for determining the usefulness of a network of email addresses?
- Which graph features or attributes best provide a means to accurately classify a graph of email addresses into a social network?

- What classification algorithms yield the best results?

## 1.2 Contributions

Previous research has shown that looking at clusters of email addresses based on their locality on a device effectively separates email addresses into networks of email addresses of similar categories [3]. Our approach directly accesses the underlying data on a device bypassing a system's interface. With the email addresses extracted along with their byte-offset location alone, we form networks that can be classified based on their usefulness. We found through testing, that Naïve Bayes and SVM were successful in identifying 100% and 95.5%, respectively of all graphs that contained useful email addresses both with areas under the curve (AUCs) above .97 and F1 scores at .80 and .90 for Naïve Bayes and SVM, respectively. Our results show that using network science metrics as attributes to classify graphs of email addresses based on the graph's topology could be an effective and efficient tool for automatically delivering evidence to an analyst.

Our approach is operating system and file system agnostic, meaning that we are not concerned with the operating system or file system that is being used on a device. We use bulk_extractor which scans an image and extracts email addresses and records its location without having to parse the file system [5]. We cluster groups of email addresses that are stored on the device in close proximity to each other and create graphs based on their connections to each other. We analyze those graphs' attributes to determine if they resemble social networks without having to look at the individual email addresses.

## 1.3 Thesis Structure

The thesis is organized as follows. Chapter 2 provides background information on topics relevant to our research, including concepts in Network Science, Graph Theory, and Machine Learning. It also provides an overview of the tools we used to extract email addresses, and the tools used to analyze the resultant graphs. Chapter 3 contains related work in digital forensics, classification research, and work done in community detection. Our methodology is detailed in Chapter 4 and Chapter 5 provides the results and analysis of research. Chapter 6 describes our conclusions as well as future work possibilities.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 2: Background

This chapter introduces key concepts relevant to our research. The concepts come from four fields of study: Digital Forensics, Network Science, Graph Theory, and Machine Learning. We use Digital Forensics to extract email addresses from a secondary storage device, Network Science to model groups of email addresses into graphs, Graph Theory to analyze the graphs, and Machine Learning to classify the groups into classes of being useful or not based. A "useful" group in our research means a group that contains email addresses that the user(s) of the device communicate(s) with.

This chapter begins with a description of Network Science, next defines graphs and their different types, and then describes different graph attributes. Next, we provide an overview of different Machine Learning strategies used for classification and provide a description of performance measures in testing learning algorithms. We then provide a description of the different tools used to extract email addresses, create graphs, visualize graphs, analyze graphs, and classify graphs. This chapter concludes with a basic overview of email protocols.

## 2.1 Network Science

Network Science is a relatively new field of study that has emerged at the end of the 20th century and has become increasingly relevant in understanding the complex behavior of real-world networks [1]. Networks have been studied for more than the last 20 years, but with the advancements in computer processing and digital storage in the era of "Big Data," we now have the unprecedented ability to map complex networks consisting of millions of objects and the connections between them [6].

As more and more complex networks were studied, common characteristics between different types of networks began appearing. Barabasi points out that "a key discovery of Network Science is that the architecture of networks emerging in various domains of science, nature, and technology are similar to each other, a consequence of being governed by the same organizing principles. Consequently we can use a common set of mathematical tools to explore these systems" [6]. This common set of tools comes from Graph Theory.

The first known paper on graphs came from Leonhard Euler in 1736 in the "Seven Bridges of Königsberg" [7], where he first introduced the notion of vertices and edges when he came up with a solution to the question of the day: "Is it possible to cross the seven bridges in a continuous walk without recrossing any of them" [7]? He used uppercase letters (vertices) to represent the pieces of land that the rivers separated and lowercase letters (edges) to represent the bridges when describing his solution. This is generally understood to be the start of Graph Theory [6], [8].

### 2.1.1 Graphs

Chartrand defines a graph in the following way: "a *graph G* consists of a finite nonempty set $N$ of objects called *nodes* and a set $E$ of 2-element subsets of $N$ called *edges*" [9]. The terms *graph* and *network* are used interchangeably throughout the Network Science community, however there is a slight difference in that a network is a graph that contains additional information [6]. The terms *node* and *vertex* are also used interchangeably with node generally used when discussing networks and vertex used when discussing graphs. We will use the term nodes throughout our paper, but will use both network and graph depending on the context.

A graph can be *labeled* or *unlabeled* with respect to its nodes and/or edges although it is more common for nodes to be labeled. A labeled graph can be enumerated and represented as a finite series of nodes and edges [10], [11]. Graphs can be either *directed* or *undirected*. In directed graphs, each node has an in-degree and an out-degree, corresponding to the number of incoming and outgoing edges, respectively, that node has [9]. A graph can be *weighted* or *unweighted*. A weighted graph is a connected graph in which edges are assigned a numeric value (called the weight of the edge). Weights can be used to emphasize that particular relationships are more important, or that a connection between two nodes has occurred multiple times. In an unweighted graph, all weights are one [9].

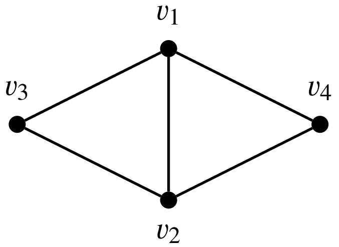
Nodes that are connected to each other by an edge are called *neighbors* and are referred to as being *adjacent*. A *path* between nodes $i, j$ in a graph as defined by Chartrand is a sequence of nodes $\{i, i+1, \ldots, j\}$ such that every consecutive pair of nodes in the sequence is adjacent. The *distance* between two nodes is the length of the shortest possible path between those two nodes in the graph [9]. The longest distance for a path between any of the pairs of nodes in a graph is the graph's *diameter* [9].

A graph $H$ is a *subgraph* of graph $G$ if its set of nodes $N(H)$ and set of edges $E(H)$ are subsets of $N(G)$ and $E(G)$, respectively. A *subgraph* $H$ is connected if for every pair of nodes $i, j$ in $N(H)$, there exists a path from node $i$ to node $j$ [9].

Graphs can be represented by pictures, formulas, edge lists, incidence matrices, or adjacency matrices. The adjacency matrix of a graph $G$ is the $n \times n$ matrix $A = [a_{ij}]$, where

$$a_{ij} = \begin{cases} 1, & \text{if } v_i v_j \in \text{E(G)} \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

For weighted graphs, $a_{ij}$ will take on the edge's assigned weight. Figure 2.1 depicts an unweighted graph $G$ and its corresponding adjacency matrix $A$.



$$A = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.1. Graph with associated adjacency matrix.

**Social Networks**

Social Networks are a specific type of network and are the focus of our research. Leskovec et al. define social networks as "graphs in which the nodes represent underlying social entities and the edges represent some sort of interaction between pairs of nodes" [2]. Examples of social networks include the relationships between a group of students in the mathematics department at the same university, the business relationships between competing insurance companies, or a family tree that goes back several generations. Each social network has its own underlying structure, however many social networks share common features. Social networks generally follow the small-world phenomenon, meaning there is a small average distance between all of nodes, and have a higher tendency for their nodes to cluster.

Furthermore, social networks usually display a power law degree distribution [1], [2], which is described in Section 2.2.2.

## 2.2 Graph Attributes

Graph attributes are often used to analyze the structure and behavior of networks. Certain types of networks often have attribute values that tend to fall within certain ranges and those values can be used to classify networks. Attribute values are not always numeric. They can be descriptive as well. Individual nodes as well as edges of graphs also have their own attributes that can be calculated. This section provides descriptions of several graph attributes as well as descriptions of attributes for nodes of graphs.

### 2.2.1 Basic Graph Attributes

The number of nodes in a graph $G$ is referred to as the *order of G*, while the number of edges is referred to as the *size of G*. Graphs' order and size can range from small graphs as seen in Figure 2.1 with 4 nodes and 5 edges, to extremely large where the number of nodes and edges can be in the billions such as a network of the neurons in the brain and their connections.

A graph's *density* is the ratio of the actual number of edges to the number of possible edges a graph can have. A *complete graph* is a graph in which every node is connected by an edge to every other node in the graph and is the most dense a graph can be. The number of edges in a complete graph is the *(number of nodes)*×*(number of nodes-1)*. A *clique* in a graph is a complete subgraph. The formula for an undirected graph's density is as follows:

$$Density = \frac{2 \times \textit{number of edges}}{\textit{(number of nodes)}\times\textit{(number of nodes-1)}}. \tag{2.2}$$

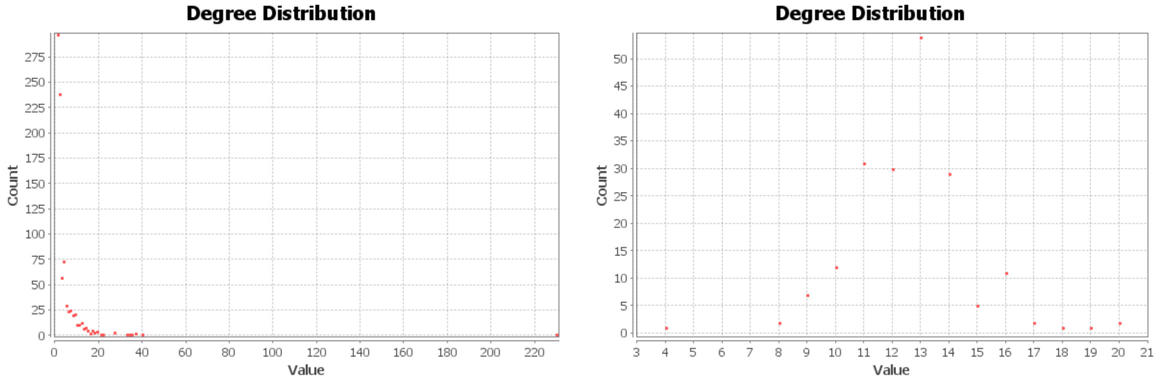
Density values range between 0 and 1, with values closer to 1 indicating that a graph is more dense, compared to sparse graphs with values closer to 0.

A measure related to a graph's density is the graph's *average degree*. A graph's average degree also measures the number of edges compared to the number of nodes, except it calculates the ratio of the number edges to the number of actual nodes. Because each edge in an undirected graph is connected to two nodes, we double the number of edges and the formula for average degree is as follows:

$$AverageDegree = \frac{2 \times number\ of\ edges}{number\ of\ nodes}. \tag{2.3}$$

### 2.2.2 Degree Distribution

An important attribute of an individual node in a graph is the number of neighbors it has, which is referred to as the node's *degree*. A graph's degree distribution is the distribution representing the frequency with which its nodes have a given degree over the network. As mentioned, social networks' degree distributions tend to follow the power-law, meaning that few nodes have a high degree while the majority of nodes have a low degree [12]. This can be seen in the left table of Figure 2.2 compared to the table on the right, which appears to follow a normal distribution.



(a) Degree distribution of a social network.

(b) Degree distribution of another subgraph from the same storage device.

Figure 2.2. Power Law Degree Distribution vs Normal Degree Distribution. The distribution on the left represents a social network and has one node out to the right with a degree of 230, and 300 nodes with a degree of one. In contrast on the right is a distribution of nodes belonging to a non-social network. The distribution of the non-social network closely follows a normal distribution with the majority of nodes, 55 having a value of 13, with a few nodes having degrees at the tails of the distribution.

**Matchings**

A *matching* is a subgraph of $G$ in which no two edges from $G$ are adjacent to each other. Necessarily, each node in a matching will have a degree of 0 or 1. A *maximal matching* is the set of edges that make up the largest matching of a graph [9]. Figure 2.3 shows graph $G$ with all 3 of its matchings below. $M1$ is the maximal matching as it is the subgraph with the most number of edges and its size is 2.

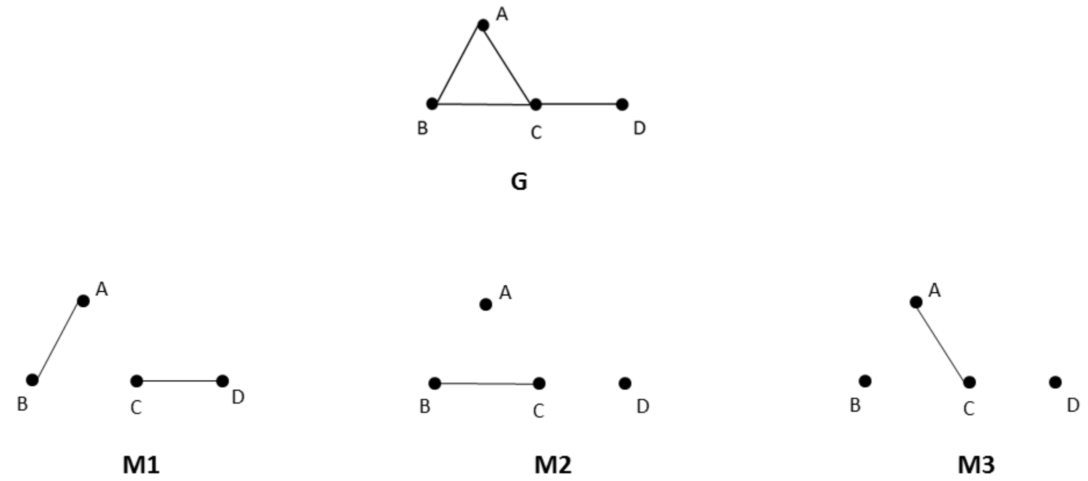


Figure 2.3. Graph $G$ with its 3 possible matchings, $M1$, $M2$, and $M3$.

### 2.2.3 Communities

Graphs representing real networks often contain communities of nodes. Barabasi formally defines a community as a "locally dense connected subgraph in a network" [13]. Intuitively, a community consists of nodes that are connected to more nodes within their community than with nodes outside of their community [1], [12]. Social networks tend to have a very pronounced community structure [1], [12], [13].

**Modularity**

*Modularity* measures the quality of communities that form within a graph. The higher the modularity value, the more likely that communities will be formed based on the graph's topology. Newman defines "the quantity $Q$ modularity as a measure of the extent to which like is connected to like in a network. It is strictly less than 1 and strictly greater than -1, takes positive values if there are more edges between vertices (nodes) of the same type than we would expect by chance, and negative ones if there are less" [1]. The formula for modularity is as follows:

$$Q = \frac{1}{2m} \sum_{i,j} [A_{ij} - \frac{k_i k_j}{2m}] \delta(c_i, c_j), \tag{2.4}$$

where $A_{ij}$ represents the weight of the edge between $i$ and $j$, $k_i = \sum_j A_{i,j}$ is the sum of the weights of the edges incident to node $i$, $c_i$ is the community to which node $i$ is assigned, the $\delta$-function $\delta(u, v)$ is 1 if $u = v$ and 0 otherwise and $m = \frac{1}{2} \sum_{ij} A_{i,j}$ [14]. A higher (closer to 1) modularity gives a stronger indication that there exists communities of like nodes. A lower modularity value indicates the opposite, revealing a lack of community structure within the graph.

### Conductance

*Conductance* measures how well the members of a community fit together. Conductance $\phi$ of a set of nodes, $S$ is defined by Leskovec et al. [2] as follows: Let $G = (V, E)$ denote a graph, and let $S \subset V$. Let $v$ be the sum of degrees of the nodes in $S$, and let $s$ be the number of edges with one endpoint in $S$ and one endpoint in $\bar{S}$, where $\bar{S}$ denotes the complement of $S$. Then, the conductance of $S$ is $\phi = s/v$, or equivalently $\phi = s/(s + 2e)$, where $e$ is the number of edges with both endpoints in $S$. In contrast to modularity, a lower conductance value will indicate a better quality community, with nodes that are more densely linked inside and sparsely linked outside [2]. The values for conductance range between 0 and 1.

### Clustering Coefficients

The clustering coefficient of a graph measures the average probability that two neighbors of a node are themselves neighbors [1]. Social networks often show a tendency for link formation between neighboring nodes when compared to random networks. This tendency is called clustering and these nodes often form tightly connected communities. The *global clustering coefficient* or *transitivity* of a graph is the fraction of 3× the number of triangles in a graph divided by the number of 3-connected nodes of a graph [15]. A *triangle* is a complete graph consisting of 3 nodes.

An alternative to the global clustering coefficient is the *average clustering coefficient*. To compute this value, the local clustering coefficient is first computed for each node. In an unweighted graph, the local clustering coefficient is the ratio of triangles through that node to the maximum number of triangles that a node can potentially be part of without altering its degree. For weighted graphs, the local clustering coefficient for a given node *i* is the geometric average of the graph's edge weights and the formula is as follows:

$$C_i = \frac{1}{deg(i)(deg(i)-1)} \sum_{j,k} (\hat{w}_{ij}\hat{w}_{ik}\hat{w}_{kj})^{1/3}, \tag{2.5}$$

where j and k represent nodes adjacent to i and $\hat{w}_{ij}$, $\hat{w}_{ik}$, $\hat{w}_{kj}$ represent the normalized edge weights between the nodes. Each edge weight is normalized by the maximum weight *max(w)* in the graph. i.e., $\hat{w}_{ij}$=$w_{ij}/max(w)$ [16].

The local clustering coefficients for all nodes are summed and the total is divided by the graph's order to get the average clustering coefficient ($C$) for the graph as follows:

$$C = \frac{1}{n} \sum_{i} C_i. \tag{2.6}$$

The global clustering coefficient and average clustering coefficient values for a graph range between 0 and 1, with values closer to 0 indicating that a smaller number of its nodes on average have neighbors that are connected. Values closer to 1 indicate the opposite, revealing a higher number of connected neighbors. The latter is often the case in typical social networks, as the friends of friends are more likely to know each other [16].

### 2.2.4 Assortativity

*Assortativity* is the tendency of nodes that are of a similar type to connect to each other [1]. For example, papers in a citation network tend to cite papers in the same field, creating a network with high assortativity. Highly assortative graphs tend to remain connected when a node with a high degree is removed. Conversely, in disassortative graphs, removal of a highly connected node will result in a highly disconnected graph as there are no redundant nodes to maintain the connections between the nodes with low degrees [17], [18].

When determining the extent of similarity between nodes in a graph, there are two types: *structural equivalence* and *regular equivalence* [1]. “Two nodes are structurally equivalent if they share many neighbors. Two nodes are regularly equivalent if they have neighbors who are themselves similar” [1]. In other words, two nodes that are regularly equivalent play a similar function in the network. They may not have the same neighbors, but many of their neighbors are similar. An example is two army squad leaders: they are regularly equivalent in that they are both connected to nodes representing squad members (though not the same squad members) and structurally equivalent in that they share the same platoon leaders and chain of command. The Pearson Correlation Coefficient is one of the most commonly used methods to measure structural similarity [1]. It is the actual number of common neighbors minus the normalized expected common neighbors. The formula for the Pearson correlation coefficient $r$ is as follows:

$$r = \frac{\sum_{xy} xy(e_{xy} - a_x b_y)}{\sigma_a \sigma_b}, \tag{2.7}$$

where $e_{xy}$ is a matrix of the fraction of all edges in the network that join together nodes with values $x$ and $y$ and $a_x$ and $b_y$ are, respectively, the fraction of edges that start and end at nodes with values $x$ and $y$. $\sigma_a$ and $\sigma_b$ are the standard deviations of the distributions $a_x$ and $b_y$ [18] .

The Pearson correlation coefficient $r$ ranges from -1 to 1 with more assortative networks being closer to 1 and disassortative networks being closer to -1.

### Graph Kernels

Graph kernels are generally used to determine the similarity between graphs, but can be used in determining the similarity between nodes as well. There are several different graph kernels and each one looks at a specific characteristic of the graphs. Computing graph kernels is generally computationally expensive as they look at specific behavioral features such as random walks, paths, cyclic patterns, or trees to determine the similarity between graphs [19]–[21].

### 2.2.5 Centrality

Centrality measures the importance of a node in a network [22]. Because centrality is computed for an individual node, some statistical measure such as the mean, median, minimum, or maximum of the nodes' centralities is needed to characterize a graph. There are different measures of centrality that prioritize different properties, some of which are more important in some networks than others. However, for each centrality measure, its values range between 0 and 1, with values closer to 1 indicating that a node is more important [22].

We now present a few common centralities.

**Degree Centrality**

The simplest centrality is the *degree centrality* of a node, which is the fraction of nodes it is connected to compared to the size of the graph [1]. The degree centrality $DC_i$ of node $i$ is as follows:

$$DC_i = \frac{deg(i)}{\sum_j deg(j)}, \tag{2.8}$$

where $j$ is a node. This centrality is computationally efficient as each node's degree is found in the graph's adjacency matrix. In an undirected graph, the sum of all the nodes' degrees is simply twice the number of edges.

**Closeness Centrality**

Freeman [23] defines the *closeness centrality* of a node as the reciprocal of the sum of the shortest path distances from that node to all other nodes. The formula for the closeness centrality $CC_i$ of node $i$ is as follows:

$$CC_i = \frac{1}{\sum_j d(i,j)}. \tag{2.9}$$

When normalized so that the maximum will always be 1 regardless of graph size, the formula is:

$$CC_i = \frac{n-1}{\sum_j d(i,j)}, \tag{2.10}$$

where $d(i,j)$ represents the distance between nodes $i$ and $j$. [23] For disconnected graphs, assuming the distance between unconnected nodes is infinity and also assuming that $\frac{1}{\infty} = 0$, the above formula will result in 0. However, we focus on connected graphs in our research so the above closeness centrality formula will suffice.

**Betweenness Centrality**

*Betweenness centrality* measures the importance of a node by determining how much traffic passes through that node when taking the shortest path between two other nodes [22]. Thus, a node with a higher betweenness centrality sees more traffic and serves as an important role in the network's connectivity. Brandes and Ulrik [22] define the betweenness centrality $BC_i$ of node $i$ as the sum of the fraction of all pairs' shortest paths that pass through that node with the formula as follows:

$$BC_i = \sum_{j,k} \frac{\sigma(j,k|i)}{\sigma(j,k)}, \tag{2.11}$$

where $j$ and $k$ are nodes, $\sigma(j,k)$ is the number of shortest $(j,k)$-paths, and $\sigma(j,k|i)$ is the number of those paths passing through some node $i$ [22].

**Eigenvector Centrality**

*Eigenvector centrality* can be thought of as an extension of degree centrality, in that it uses weighted degrees to compute a node's centrality. However, eigenvector centrality takes into account the centrality of its neighbors as well. If a node's neighbors are important, then that node is important. The eigenvector centrality $x_i$ for a node $i$ is computed recursively by making $x_i$ proportional to the average of the centralities of its neighbors [24].

### 2.2.6 Normalization

The above attributes produce values in different ranges. For example, the average clustering coefficient produces values between 0 and 1, while the modularity of a graph produce values between -1 and 1. The number of nodes and edges in a graph varies greatly as well with values from the tens and hundreds to values in the millions. Often, it is beneficial to normalize the values of different characteristics when they are being compared to each other. Two such normalization methods defined by Li et al. [25] are range and z-normalization. In range normalization, each value $x$ of a characteristic is transformed into $r(x) = \frac{x-min}{max-min}$, where min and max denote the minimum and maximum values of the characteristic. In z-normalization, $x$ is replaced by its $z\text{-}score(x) = \frac{x-\mu}{\sigma}$ where $\mu$ and $\sigma$ are the mean and standard deviation of that characteristic [25].

## 2.3 Classification

Classification is the problem of identifying to which set of categories an observation belongs. Two major approaches to classification problems are supervised and unsupervised learning. In supervised learning, the classifier is first trained on a dataset for which the correct classification labels are known. The goal is to use this training set of correctly identified observations to construct a good and useful approximation model that can detect patterns and make predictions about data for which there is no prior knowledge [26], [27]. Unsupervised learning, in contrast to supervised learning, draws inferences about datasets without having correctly identified observations available. "Unsupervised learning methods are often used in bioinformatics for sequence analysis and genetic clustering; in data mining for sequence and pattern mining; in medical imaging for image segmentation; and in computer vision for object recognition" [28].

### 2.3.1 Decision Trees

Decision tree induction is a simple and effective form of machine learning. It takes an input vector of attributes and returns an output. The decision tree is a series of tests that represent a function. The input and output values can be binary, multiclass, numeric, or a combination of the three. The decision tree reaches a decision by running the function. Each node in the tree corresponds to a test in the function. The result of a test determines which test will be performed next. If the decision tree is set up correctly, the tests will eventually reach a decision at a leaf node [27]. Figure 2.4 shows a simple boolean classification problem of deciding whether or not to buy a home.

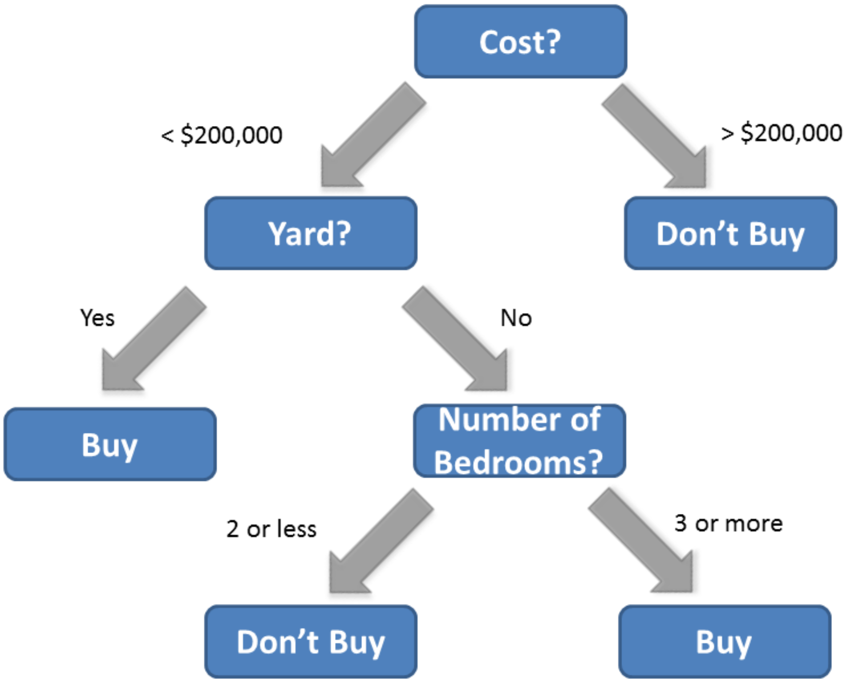


Figure 2.4. Decision strategy to buy a home. A decision to buy or not is reached at each leaf node based on the home's cost, whether or not it has a yard, and the number of bedrooms.

Classification problems are often not as straightforward as shown in Figure 2.4. Most real world data will have some amount of *noise*. Noise occurs for several reasons. There could be imprecisions in the recording of input attributes. The original classification of the training set may be flawed, resulting in errors that could be carried throughout the rest of the experiment. Also, not all datasets act in a consistent way that can be predicted [26].

### 2.3.2 Bayesian filters

Bayes' rule can be used in classification problems to develop learning algorithms in supervised learning. The goal is to approximate the function that maps inputs into outputs, or equivalently, determine the probability of potential outputs given a particular set of inputs, that is $P(Y|X)$ where $Y$ is the output and $X$ is the input. By Bayes' rule, we know $P(Y|X)P(X) = P(X|Y)P(Y)$. We can therefore calculate $P(Y|X)$ by using a training set to estimate the total probability of the output $P(Y)$ and the probability $P(X|Y)$ of the relevant input, given that output. The approach is to build a Bayesian classifier from training data by estimating the probability that a certain output, $P(Y)$ will occur, and by estimating the probability that the input occurred when we know the output $P(X|Y)$. Once we have estimates, we can apply Bayes rule, referenced below to compute the probability of an output given a specific set of inputs. Bayes rule is as follows:

$$P(Y = y_i|X = x_k) = \frac{P(X = x_k|Y = y_i)P(Y = y_i)}{\sum_j [P(X = x_k|Y = y_j)P(Y = y_j)]}, \tag{2.12}$$

where $y_m$ denotes the $m$th possible value for $Y$, $x_k$ denotes the $k$th possible vector value for $X$, and the summation is over all possible values of the output [29]. Determining an accurate estimate of the set of probabilities of the different outputs $P(Y)$ can be achieved with a relatively small training set. Finding an estimate for the possible inputs given knowledge of the output $P(X|Y)$ is much more difficult. In the case where the output is boolean (true or false), and the input is made up of $n$ different attributes, we need to estimate a set of parameters for each possible input $2^n$ times. For 30 different inputs, 3 billion estimates of parameters would be required [26], [27], [29].

In Naïve Bayes models, each attribute is treated as if it is *conditionally independent* of each other, meaning that if an event happens in which $A$ and $B$ are any attributes for that event, knowledge of $A$ occurring provides no information on the probability of $B$ occurring, and knowledge of $B$ occurring, provides no information on the probability of $A$ occurring [30]. Equivalently, in mathematical notation:

$$P(X_1 ... X_n|Y) = \prod_{i=1}^{n} P(X_i|Y). \tag{2.13}$$

Assuming conditional independence greatly reduces the number of parameters that need to be estimated down to just $2n$ when assuming boolean values as we did above [29].

The Naïve Bayes approach can be used when the input is made up of continuous values. A common approach is to assume that for each possible discrete output value, that the distribution for each input is Gaussian, which has a mean and standard deviation specific to that output. In the continuous input case, estimates of the mean and standard deviation are made for each input, but they are still assumed to be independent so the reduction in computation time is still achieved.

### 2.3.3 Linear Regression

Linear regression in machine learning is another approach to predict the behavior of models on a set of inputs that result in an output. The approach assumes a linear relationship between the input variables and an output variable, with the output variable being predicted from a linear combination of the input variables. Based on a training data set, each input is assigned a scaled value that is reflective of that input variable's effect on the resulting output variable. In a simple linear regression problem, with a single input $x$ and an a single output $y$, the model would be [27]:

$$y = \beta_\circ + \beta_1 x, \tag{2.14}$$

where $\beta_1$ is the assigned scaled value for the input variable $x$ and $\beta_\circ$ is the intercept. Models with multiple input variables, $x_1$ thru $x_n$ can be represented as follows:

$$y = \beta_\circ + \beta_1 x_1 + ... + \beta_n x_n. \tag{2.15}$$

From the training set, we know $y$ and $x_1$ thru $x_n$ for each trial and then estimate the $\beta$ values using various statistical methods. The $\beta$ values form the model and are used to estimate the output when given a set of inputs for trials we are trying to predict [27].

### 2.3.4 Logistic Regression

Logistic Regression differs from linear regression in that logistic regression predicts the probability of whether the output will be in a specific class or not, where linear regression predicts a continuous output value based on a set of inputs. Logistic regression is a special case of linear regression using the Bernoulli distribution in which the output variable is binary (either in a class or not) with a probability between 0 and 1. In order to restrict the values between 0 and 1, the *logistic transformation* function $log\frac{p}{1-p}$ is used as a link function to satisfy the continuous requirement and then used to conduct linear regression. The resulting logistic regression model is as follows [26]:

$$log\frac{P(Y)}{1-P(Y)} = \beta_\circ + \beta_1 x_1 + ... + \beta_n x_n, \tag{2.16}$$

where the $\beta$ values are the estimates that are calculated from the training set for each input variable $x_i$. Solving for $P(Y)$ gives:

$$P(Y) = \frac{1}{1+e^{-(\beta_\circ+\beta_1 x_1+...+\beta_n x_n)}}. \tag{2.17}$$

$P(Y)$ is the estimation whether the result will fall in a given class.

The $\beta$ values are often calculated using maximum likelihood estimation (MLE). MLE takes the mean and variance parameters from the training set and attempts to find the $\beta$ values that make the observed results the most probable given the model. Once the $\beta$ values are computed, the logistic regression model is used to predict the probability of the output belonging to a class [29].

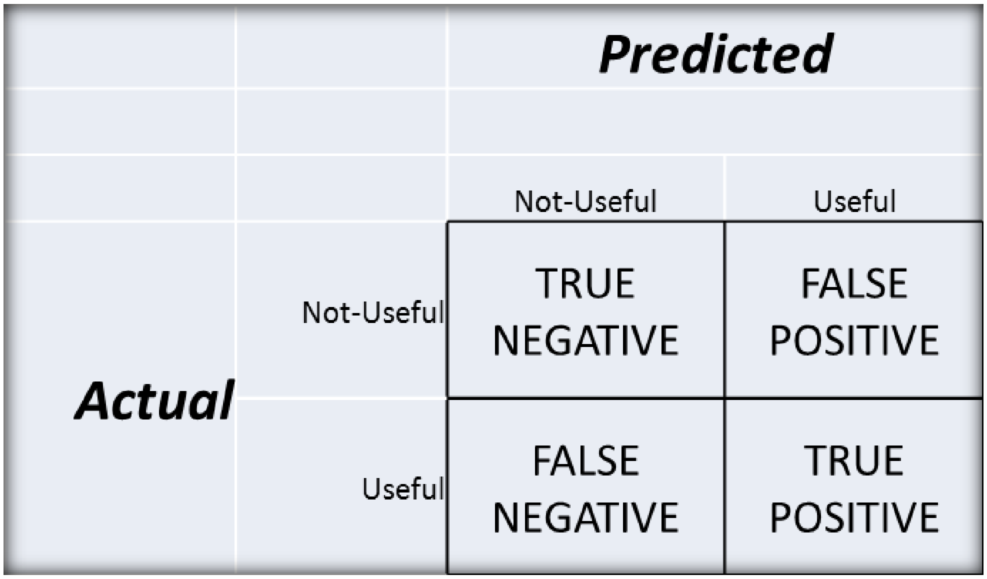
### 2.3.5 Support Vector Machines

Support vector machines (SVM) are another type of supervised learning model. Given a set of training examples, with the knowledge of which class each belongs to, the SVM training algorithm attempts to classify new instances into a category based on the model created from the test data. The model non-linearly maps input vectors to a very high-dimension feature space which is used to formulate decisions of class membership [31].

### 2.3.6 Testing Performance of Learning Algorithms

The F1 and AUC are two commonly used scores that measure the performance of learning algorithms. Both values can be computed from values that are found in a confusion matrix. In a confusion matrix, the actual number of class instances is shown with the number of predicted instances from the learning algorithm for each class. Figure 2.5 shows a confusion matrix example for a notional learning algorithm that classifies an instance into two classes:

Useful or Not-Useful.



|  |  | Predicted |  |
|---|---|---|---|
|  |  | Not-Useful | Useful |
| Actual | Not-Useful | TRUE NEGATIVE | FALSE POSITIVE |
|  | Useful | FALSE NEGATIVE | TRUE POSITIVE |

Figure 2.5. Confusion Matrix for two classes: Useful or Not-Useful.

The value in the true negatives box of Figure 2.5 represents those instances that are Not-Useful and correctly predicted as such. In the lower-right corner are the true positives: those instances which are Useful and accurately predicted as being Useful. The upper right box shows the number of false positive instances. These are actual Not-Useful but wrongly predicted as being Useful. The false negatives are in the lower-left box and are the instances that are Useful but wrongly predicted as being Not-Useful.

Precision gives a fraction of how many of the instances that are predicted as being relevant are actually relevant. Recall is the fraction of instances that are relevant and accurately predicted as being relevant. The F1 score takes into account both precision and recall. The formulas are as follows:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}. \tag{2.18}$$

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}. \tag{2.19}$$

$$F1Score = 2x\frac{precision\ x\ recall}{precision + recall}. \tag{2.20}$$

The AUC looks at the receiver operating characteristic (ROC) curve. The ROC curve is generated by plotting the true positive rate (recall) on the y-axis and the false positive rate on the x-axis. The false positive rate is how often the classifier predicts an instance as being relevant when it is not actually relevant [32]. A classifier that performs no better than random guessing has a curve that is close to a straight line and along the diagonal with a slope of 1. The AUC for such a classifier is .5, where a classifier that can effectively separate classes will have ROC curve that closely hugs the upper left corner of the graph whose AUC value will be closer to 1. Figure 2.6 compares 3 different ROC curves.

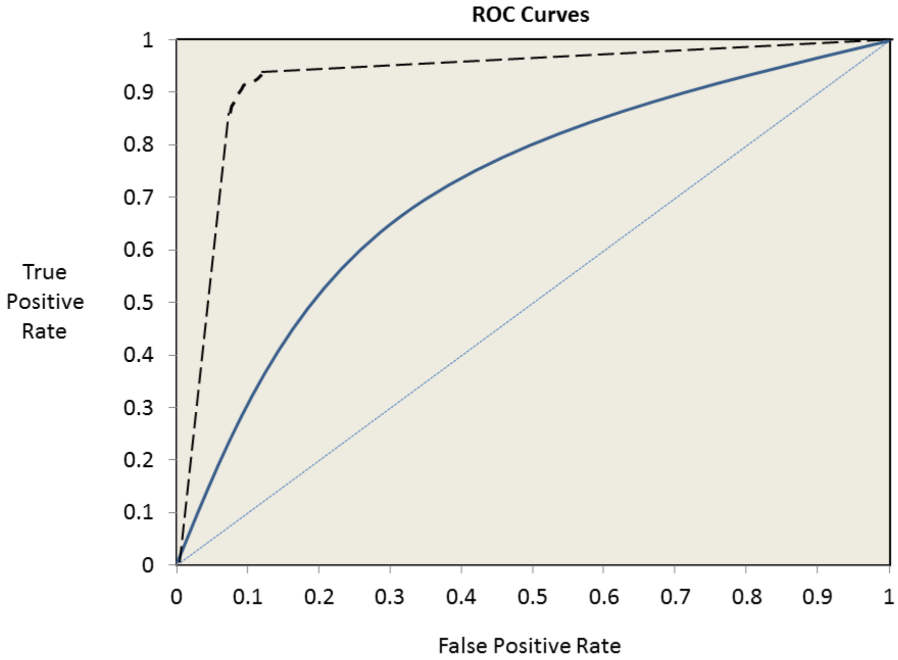


Figure 2.6. ROC curves compared. The dashed black curve at the top has the highest AUC. The dotted blue curve that is a straight line has the lowest AUC at 0.5 indicating performance no better than random guessing. The solid blue curve in the middle is representative of a good AUC.

## 2.4 Tools

With the growth of networks and the need to understand them, we require tools capable of performing complex computational analysis. The primary tools used throughout our research. The tools are open source and widely used throughout the Network Science and digital forensics community.

### 2.4.1 bulk_extractor

*bulk_extractor* is a tool that extracts useful information from a device. The first prototype was developed between 2003 and 2005 as a bulk media analysis tool to assist in investigations. bulk_extractor can be used to extract objects such as email addresses, credit card numbers, URLs, and other types of information from digital media [5]. A key feature of bulk_extractor is that it ignores file system structure, allowing it to scan different sections of the disk in parallel, and also allowing it to scan any type of digital media. It has been used to process "hard drives, SSDs, optical media, camera cards, cell phones, network packet dumps, and other kinds of digital information" [5]. In our research, the useful information that is extracted are objects that "look"[1] like email addresses.

When a disk image is scanned using bulk_extractor, the useful information is stored in *feature* files [5]. The feature files can be then be further analyzed and processed. A histogram is also created showing the frequency of the objects of a given media device. This is beneficial because the frequency with which email addresses, phone numbers, credit card numbers, or other important artifacts appear is often a useful indicator of importance in investigations [5]. The byte offset where an object is stored is recorded and saved to the feature file, and can be used for further analysis.

As discussed in Section 2.5, email addresses from a variety of different sources are stored on a device. For example, there are over 20,000 Linux developers whose email addresses, when used in the context of the software, aren't of significance when conducting analysis. In order to filter out email addresses that have no value to an investigation, bulk_extractor has an option to use context-sensitive stoplists that ignores those email addresses in program binaries, but will not ignore the same email addresses in email messages, in case they are

[1] Email addresses follow the format someusername@somedomain.some_tld; just because an object may be in an email address format does not always mean it is an actual email address.

actually used for communication [5].

### 2.4.2 Betographer

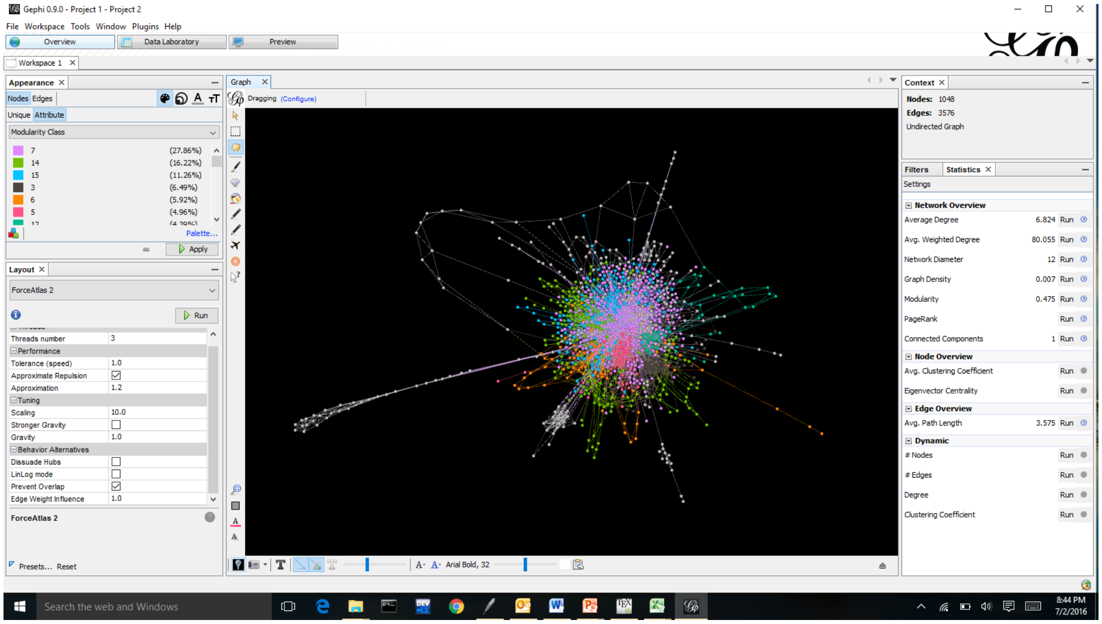
The feature files created by bulk_extractor are .txt files. In order to convert the .txt files into a format that can be understood by graph analysis tools, a Python script, Betographer.py was written while conducting prior research. Betographer can create a graph file (.gexf) for the entire feature file, but also has the option to create a smaller graph files of connected components of the graph [3]. Betographer also creates a summary file with basic statistics of the graph files to include the size and order of the created graph files.

### 2.4.3 Gephi

From Bastian, "Gephi is an open source software for graph and network analysis. It uses a 3D render engine to display large networks in real-time and to speed up the exploration" [33]. Gephi contains modules that can import, visualize, filter, manipulate, and export all types and sizes of networks. It is built on a multi-task model, allowing it to take it advantage of multi-core processors. This allows Gephi to handle large networks, over 20,000 nodes, although networks of this size do slow down Gephi's functions. Gephi has a variety of visualization algorithms whose parameters can be adjusted to highlight the features or structure of a graph as desired. Nodes, edges, and their characteristics can be viewed in spreadsheet format with the option to sort, add and remove nodes, label, and import or export files. Gephi computes many graph statistics, and can adjust the size and/or color of the nodes and edges based on the values of those statistics [33]. Figure 2.7 shows a screenshot of the Gephi interface.

### 2.4.4 NetworkX

NetworkX is a Python language package that can be used in exploring and analyzing networks through the built-in network algorithms. The core package provides data structures capable of representing many types and different sizes of networks [34]. NetworkX is designed to handle data on a scale relevant to the size of today's networks. Its core algorithms rely on extremely efficient code and can handle graphs with 10s of million of nodes and 100s of million edges [35]. Schult and Swart [34] describe that once a network is represented as a NetworkX object, the network structure can be analyzed using



Modularity Class

| | | |
|---|---|---|
| 🟪 | 7 | (27.86%) |
| 🟩 | 14 | (16.22%) |
| 🟦 | 15 | (11.26%) |
| ⬛ | 3 | (6.49%) |
| 🟧 | 6 | (5.92%) |
| 🟥 | 5 | (4.96%) |
| 🟩 | 12 | (4.39%) |

ForceAtlas 2

| | |
|---|---|
| Threads number | 3 |
| ⊟ Performance | |
| Tolerance (speed) | 1.0 |
| Approximate Repulsion | ☑ |
| Approximation | 1.2 |
| ⊟ Tuning | |
| Scaling | 10.0 |
| Stronger Gravity | □ |
| Gravity | 1.0 |
| ⊟ Behavior Alternatives | |
| Dissuade Hubs | □ |
| LinLog mode | □ |
| Prevent Overlap | ☑ |
| Edge Weight Influence | 1.0 |

ForceAtlas 2

Context

**Nodes:** 1048
**Edges:** 3576
Undirected Graph

Filters Statistics

Settings

| | | |
|---|---|---|
| ⊟ **Network Overview** | | |
| Average Degree | 6.824 | Run |
| Avg. Weighted Degree | 80.055 | Run |
| Network Diameter | 12 | Run |
| Graph Density | 0.007 | Run |
| Modularity | 0.475 | Run |
| PageRank | | Run |
| Connected Components | 1 | Run |
| ⊟ **Node Overview** | | |
| Avg. Clustering Coefficient | | Run |
| Eigenvector Centrality | | Run |
| ⊟ **Edge Overview** | | |
| Avg. Path Length | 3.575 | Run |
| ⊟ **Dynamic** | | |
| # Nodes | | Run |
| # Edges | | Run |
| Degree | | Run |
| Clustering Coefficient | | Run |

Figure 2.7. Gephi screenshot displaying a graph in which the nodes are colored with respect to their modularity class. The nodes and edges are shown with the Force Atlas 2 visualization algorithm and some of the key graph statistics are displayed on the right.

standard algorithms to compute the network's characteristics (described in Section 2.2) such as the graph's degree distributions, clustering coefficients, and centrality measures. The NetworkX library combined with other Python packages, including SciPy, NumPy, and Matplotlib make it a valuable tool when conducting computational network analysis [34].

### 2.4.5 Orange

Orange is a software machine learning and data mining suite that is written in Python. It provides a platform for experiment selection, recommendation systems, and predictive modeling. The Orange library is a hierarchically-organized toolbox of data mining components. Data filtering and probability assessment are at the bottom of the hierarchy and are assembled into higher-level algorithms, such as classification tree learning at the top. Orange is implemented through components called widgets which range from simple data visualization to predictive modeling. Orange uses a canvas as its graphical front-end interface and widgets that are manipulated for its main functions: data management and preprocessing, classification, regression, association, clustering, evaluating, and projecting [36], [37].

Orange supports various sampling methods when testing learning algorithms. It supports cross-validation which splits data into a selected number of folds, where the learning algorithm is tested by leaving one fold out at a time. The fold that is left out is classified and this process is repeated for each fold. A similar method is the leave-one-out method where one instance, as opposed to the entire fold is held out. Orange also supports random sampling where the size of the training and testing sets can be chosen, and then repeated for a selected number of iterations [38]. Orange provides the option to stratify data meaning that the sample sets will have the same proportions for each class as the entire set. This is useful for data that has a disproportionate number of instances belonging to a certain class [39].

## 2.5 Email Properties

Our research looks at email addresses on digital storage devices to determine which email addresses may be used for communication and which email addresses are artifacts of installed software, or are other email addresses that aren't associated in communicating with the owner of the device. Thus, we provide a basic overview of email protocols and features.

POP, or Post Office Protocol, is the original email protocol and was developed in 1984. Since then, there have been upgrades. POP3 is the current version and remains one of the most popular protocols [40]. POP emails are downloaded from the server's inbox to the user's device. Email clients using POP generally connect to the Internet, retrieve emails, and store them on a local device, at which time the emails are usually deleted from the server, unless the option to leave the emails on the server is enabled. In this way, emails become tied to the specific machine that retrieved them and can be accessed when that device is not connected to the Internet.

Internet Message Access Protocol (IMAP), created in 1986, is more suitable to the always available Internet model. It allows users access to their emails as long as they are connected to the Internet. Remote email servers store the user's email addresses until the user deletes[2]

[2] Delete does not necessarily mean fully erase in this context. Once a user deletes an email from a server, it may go into a trash bin for a certain period of time before it is completely removed. Even then, the email server may still retain the email, but from the user's point of view, the email appears to be deleted.

them [40]. Users will often make local archived copies of emails using .pst files or other methods of their choosing.

Emails and their corresponding email addresses may be stored in different locations. Sometimes they are stored locally on the hard drive, while other times, they are stored on a remote web server, and sometimes a combination of the two locations. Webmail is designed to be operated over the Internet, with no additional applications or software required beyond the web browser in most cases. On the other end, email clients are programs that are installed locally, and interact with remote email servers in downloading, sending, and receiving email. Here, the local machine does more of the work in creating the user interface, whereas the browser is the interface doing all of the work in a webmail based system. The system being used determines where the email addresses will be stored into memory. Devices using email clients will generally have more email addresses stored locally on their device than webmail-based clients. This does not mean that computers using webmail-based email services will not have email addresses used for communication stored in their memory. Users can still save emails and addresses, and when a user opens or sends emails, much of that information is still saved [40].

There are a variety of different ways that email addresses end up on a given storage device. Some direct ways include Microsoft's Personal Storage Table (.pst) and Off-line Storage Table (.ost) These allow users to save their emails on their device. Similarly, a user can use Personal Address Books (PAB) and Offline Address Book (OAB) to store contact information on their devices and access it when they are not connected to the Internet. Email addresses are also stored in various file formats such as *mbox*. All messages in an mbox mailbox are stored as plain text in a single file.

In addition, email addresses are often embedded in websites and get saved to a device when certain sites are visited. Software programs often contain thousands of email addresses of developers that all get stored on a device. Databases such as SQL often have email addresses that get stored if the user uses that service on their device.

THIS PAGE INTENTIONALLY LEFT BLANK

# CHAPTER 3:
# Related Work

Searching through a device to find email addresses associated with social networks is not an easy task. Garfinkel points out that "there seems to be a widespread belief, buttressed on by portrayals in the popular media, that advanced tools and skillful practitioners can extract actionable information from practically any device that a government, private agency, or even a skillful individual might encounter" [41]. The reality is that these systems are complex and each one has its own nuances. There are several methods and tools for examining storage devices to uncover social networks. Once such networks have been established, Community Detection techniques can be used to attempt to discover networks of similar objects and group them together. This chapter presents some of those community detection methods being used today, as well as a brief summary of previous research conducted in finding and classifying communities in networks relevant to our research.

## 3.1 Identifying Social Network Artifacts with Digital Forensics

The initial triage of a device is meant to produce leads in an investigation in a timely manner. Garfinkel [41] described different processing models in conducting the initial triage. There is the "Visibility, Filter and Report" model, which looks at all the data on a device and organizes it into a tree structure, with the root or base of the tree as the point from which all other data can be reached. Extracted data can be stored as tables, filtered, and searched for specific content, for example an email address. However, this model is not able to extract and prioritize all email addresses that may be present on a device, since many email addresses may be stored at locations no longer reachable from the root node (e.g. because they were deleted). An alternative processing model is referred to as "Stream-based disk forensics" which processes an entire disk image from beginning to end as a byte-stream. This model, which we use in our research is exhaustive in scanning all data from a drive and eliminates the time spent for the drive head to seek through a device. Stochastic analysis is another model, which involves sampling and processing random sections of the drive. This method can be extremely fast but runs the risk of missing important data. When time is of the utmost concern, there are triage-oriented approaches sometimes referred to as "priority models," which produce a subset of data from the original set to examine further. The goal is to minimize the subset for the analyst to review, while not omitting any critical data [41], [42].

### 3.1.1 Email Forensic Tools

There are several tools available that assist in analyzing the source and content of emails (as opposed to email addresses only) on a given device. Depending on the task, certain email forensic tools are better suited than others in retrieving relevant information. Devendran et al. [43] compared 5 open source tools that are commonly used today: MailXaminer, Aid4Mail, Digital Forensics Framework, eMailTrackerPro, and Paraben E-Mail Examiner. These tools are fairly robust and are able to do much more than extracting user-related email addresses from devices, which is the starting point of our research. The authors looked at nine different criteria that were derived from a set of desired attributes required by forensic tools [4]. The different criteria ranged in measuring a tool's capabilities to different usability features of a tool. The criteria relevant to our research included the requirement of an input file, type of information extracted, recovery capability, and email file format supported.

For the input file requirement, the majority of the tools compared are capable of analyzing emails stored on a hard disk. One requires a separate input file, which must be produced in a separate pre-processing step. Some of the tools are able to conduct analysis both on and offline, which differs from our research as we focused only on offline devices. As mentioned, these tools do more than just extract email addresses from a device. An analyst can inspect complete emails, including the time and dates of when it was sent and received, as well as other metadata associated with emails. He or she could then attempt to create a social network based on what was extracted by the forensic tool. Our approach is different in that our intent was to discover social networks based solely off email addresses' locations on a device without the additional overhead of analyzing emails, in hopes of developing a fully automated, scalable method for constructing social networks.

All but one of the email forensic tools compared offer some sort of deleted email recovery functionality. Some are able to recover directly from the hard disk, and some are able to recover emails from the trash bin, or deleted email folders. In our research, we rely on bulk_extractor, referenced in Section 2.4.1, which ignores file structure allowing it to find email addresses regardless of where they are stored. This saves time by not having to re-create the file system structure and search for specific folders. bulk_extractor's file system agnostic approach offers another benefit: the email forensic tools are limited to extraction of supported email formats only. For example, MailXaminer supports Gmail, yahoo, Hotmail, IMAP, Mozilla Thunderbird, Lotus Notes, Outlook, Exchange, and Mac Outlook email formats [43]. If a device uses an obscure format not recognized by the tool, then those emails will most likely be ignored.

An issue with the "stream-based" approach is that it produces a very good recall at the expense of precision—that is, it extracts what is needed but brings with it a lot of noise. A stop list (see Section 2.4.1) offers a partial solution in eliminating excess noise, such as those email addresses associated with the operating system. The National Software Reference Library (NSRL) contains databases of email addresses that are associated with different software and is a useful source of addresses to create a stop list. However, as Garfinkel points out, an email address that may appear to be an artifact of the operating system could actually be relevant to an investigation [4]. To resolve this, Garfinkel et al. introduced *context-sensitive stop lists* which look at the characters surrounding the email address to determine the context of the email address. Thus, email addresses that appear to be part of the operating system are only ignored when they are in the context of the operating system [4].

### 3.1.2 Identifying Uninteresting Files

Rowe's [44] approach looks at eliminating noise, but focuses on eliminating uninteresting files from a device. Uninteresting files are defined as files which contain no information specific to the users of a drive, such as operating system and application files that do not contain user-created information. This strategy can be leveraged to in eliminate uninteresting email addresses by whitelisting addresses associated with these files. Rowe used several different heuristics to determine which files may be uninteresting when compared with files found in drives across his test corpus:

- Files with the same hash value.
- Files with the same full path.

- Files whose pair of the file name and the immediate parent directory are the same.
- Files with the same creation times within a short period.
- Files created unusually frequently in particular weeks.
- Files with the same size and extension.
- Small files.
- Contextually uninteresting files.
- Files in known uninteresting top-level or mid-level directories.
- Default directory records.
- Files with known uninteresting extensions.

The method was tested on a data corpus of over $3,400$ drives containing $83.8$ million files. First, the hash values of the files were compared with known uninteresting hash values of files from the (NSRL), resulting in the removal of 25.1% of already known uninteresting files. After filtering out the hash values of known files from the NSRL, an additional 54.7% of files were eliminated that matched two of the nine criteria from the methods, with false negatives being estimated at 0.1% and false positives at 19.0% [44].

## 3.2 Classification

The above tools are designed to extract emails from a device for an analyst to further examine. We used bulk_extractor in our research to extract email addresses from a device regardless of whether they are used for communication. We form graphs based on the email addresses' proximity to each other and we focus on the classification problem to characterize the resulting graphs. This section looks at previous work on the classification of networks.

### 3.2.1 Graph Classification Using Global Topological Attributes

Li et al. took an "approach to graph classification that looked at feature vectors constructed from different global topological attributes. Their main idea was that graphs from the same class should have similar topological attributes" [25]—this idea was at the foundation of our methodology. Their approach created a vector of 20 graph attributes to develop a model that predicted the class label for a graph. They tested their method on three commonly used datasets in graph classification: chemical compounds, proteins, and cell graphs in which they compared the results of their method to different graph kernels (referenced in Section

2.2.4) which looked at common patterns between graphs [25].

The attributes they used range from a graph's basic attributes, such as the size and order of graphs, to more computationally complex attributes, such as the average clustering coefficients, closeness centrality, and eigenvalue measures for each graph. Each graph was represented as a vector of its 20 attribute values. Attribute vectors are then classified using a support vector machine (see Section 2.3.5) [25].

The authors used range normalization and z-normalization (defined in Section 2.2.6) to prevent attributes with large values from exerting undue influence on the classification decision [25]. They tested the three datasets using non-normalized values, range normalized values, and z-normalization values. As expected, the normalized values produced more accurate results, with z-normalization generally outperforming range normalization. Their method worked better in accurately classifying chemical compounds roughly half of the time, depending on what chemical compound was being evaluated, with values ranging from 60 to 90 percent accuracy. For the protein and cell-graph datasets, the authors' method had better accuracy results than the graph kernel methods in all the trials in classifying the data [25].

The authors also measured the computation time of their method compared to the graph kernel methods. Graph kernel computations are much more involved than computing a feature for a single graph. Because of this, the authors found significant computational savings in time with their method, saving several seconds in computation time on the smaller graphs, and saving hours on some of the larger data sets [25].

Finally, Li et al. [25] performed a detailed study to determine which attributes carried the most information. They found that although the most important graph features vary depending on the data set, some general conclusions could still be made. With range normalization, the top features were: average clustering coefficient, number of nodes, number of eigenvalues, and number of edges of the graph. The top features for z-normalization were similar, and were: number of nodes, average degree, average clustering coefficient, number of edges, and the number of eigenvalues [25].

### 3.2.2 Community Structure in Large Social Networks

Leskovec et al. used “approximation algorithms for the graph partitioning problem in an attempt to characterize, as a function of size, the statistical and structural properties of partitions of graphs that could be interpreted as meaningful communities” [2]. They studied over 70 real-world networks with a variety of different sizes. They determined the quality of a community by measuring its conductance as defined in Section 2.2.3. Their main finding was that as networks grow in size, that the better communities gradually “blended in” with the rest of the network and would resemble a community less and less. In determining this, they “introduced the network community profile plot (NCP plot), which measures the quality of the best possible community in a large network as a function of the size of the community. The conductance of a set of nodes measured how well that set of nodes form a community while the shape of the NCP plot provided insight into the community structure of a graph” [2].

For the networks that they studied, the NCP plots sloped downward until the number of nodes reaches roughly 100. This indicated the lowest conductance values (and thus "best" communities) reached their max at a size of 100 nodes. After 100, the conductance of the communities grew (lowering the quality of communities) and the slope rose. These results implied that the best formed communities were relatively small. As a community grew in size, it tended to blend in with the rest of the network [2]. An extension of this idea based on Network Community Profile was studied for very large networks by Jeub et al. [45].

## 3.3 Community Detection

Detecting and identifying communities is an important aspect of Network Science. Just like a neighborhood community, communities in Network Science are often thought of as having more connections between the members of that community than with members outside. The general method in studying networks and their underlying community structure as Leskovec et al. point out is as follows:

1. Model data in graph consisting of nodes representing the objects and edges between those nodes representing some connection [2].
2. Make educated guess on what communities should form [2].
3. Determine how a node falls into its respective community [2].

4. Optimize process of determining communities [2].
5. Evaluate fitting of nodes in to communities [2].

We used a similar method in studying our networks, except that our communities were simply the largest connected components of co-located email addresses on a device. Each component represented a community that we found contained email addresses that naturally interacted more strongly amongst themselves. The following papers demonstrate different techniques for detecting communities in a variety of different networks.

### 3.3.1 Community Detection through Graph Kernels

Wang et al. looked at "community kernel detection in order to uncover hidden community structure in large social networks. They observed that influential users paid closer attention to those who were more similar to them, which led to a natural partition into different community kernels" [19]. They used three well-known social networks in their experiments. A common feature that they discovered in social networks is that they followed the Pareto principle, which is that for many phenomena, 20% of invested input is responsible for 80% of the results. For example, Wang et al. explain that "80% of a country's land is owned by 20% of its citizens, and 80% of a company's sales comes from 20% of its clients" [19]. Similarly, they explain statistics have shown that less than 1% of the Twitter users produce 50% of the content [19]. Wang et al.'s approach was to determine the underlying structure of communities of those influential users and their followers as they have the most profound impact on the community structure [19].

The authors looked at two subtasks: first, determine the key (kernel) players, and second, to discover how those communities are structured. Each kernel was closely associated with an auxiliary kernel in which members of a specific community kernel had more connections compared to other community kernels. As Wang et al. point out, "in a co-authorship network, a community kernel can be a group of senior professors in a specific research area, while its auxiliary community consists of students or junior researchers in the same research area" [19]. Their method was able to differentiate between kernel and auxiliary communities where other methods, such as modularity and conductance ignore the differences between the two types of communities. Wang et al. proposed two different algorithms to differentiate between the two in determining community detection [19].

The first was a greedy algorithm that selected a random vertex to initialize a community kernel set, and then added to that set nodes that shared the most number of connections. The routine was repeated with different nodes as the initial node to minimize the effect of random selection. The second algorithm was the Weight-Balanced Algorithm (WeBa), which assigned weights to nodes to signify their relative importance for each community kernel. A resultant vector of the weights was created for each kernel and the vectors were then maximized to determine the optimal community kernel. To find the associated auxiliary community for each node not belonging to a community, the community kernels were first ranked. That node was then placed in the highest ranked community kernel and this process was repeated until all nodes were accounted for. The resultant auxiliary community of a community kernel consisted of all nodes that belonged with that community kernel [19].

The authors measured pairs of nodes and how well they belonged to the same community kernel using precision, recall, and F1-scores. Scoring was based on the number of pairs of nodes correctly clustered into the same community kernel. The pairwise resemblance was used to measure how similar a given community kernel $C$ was to the ground truth $A$ by $\frac{A \cap C}{A \cup C}$, the Jaccard similarity of sets of pairs [19]. Wang et al. compared their algorithms with other community detection algorithms, including those based on conductance, high degree ranking, page-rank, modularity, and betweenness. For the Wikipedia and co-author datasets, the WeBa significantly outperformed the rest of the algorithms. The greedy algorithm outperformed the others as well, but was not as successful as the WeBa in accurately detecting communities. WeBa performed on average between 14-50% better whereas the greedy algorithm performed on average less than 10% worse than WeBa. Both WeBa and their greedy algorithm computation time was significantly less than the other methods [19].

### 3.3.2 Community Detection using Betweenness

Newman and Girvan described betweenness based community detection algorithms. Their techniques were “aimed at discovering natural divisions of social networks into groups, based on various metrics of similarity or strength of connections between nodes” [46]. There are two general methods when constructing communities: agglomerative, which starts with an empty list of nodes and edges, and adds pairs of nodes with the highest similarity, and divisive methods, which the authors focus on where the pairs of nodes that are the least similar have their edges removed dividing the network into smaller communities [46].

Newman and Girvan's algorithm focused on finding edges with the highest betweenness (see Section 2.2.5) which is a measure that favors edges that lie between communities and penalizes those that lie inside communities. They tested multiple betweenness discovery techniques and found that the results were similar regardless of which technique is used. The simplest method they used was to find the shortest paths between all pairs of nodes and count how many ran along each edge. Another method was the random walk approach which calculated the expected number of times that a random walk between a particular pair of nodes would pass down a particular edge. Their community detection algorithm is as follows [46]:

1. Calculate betweenness values for all edges in the network [46].
2. Remove the edge with the highest value [46].
3. Calculate betweenness for remaining edges [46].
4. Repeat process from step 2 until all edges have been removed [46].

The authors pointed out that the re-calculation step after the removal of the edge with the highest betweenness value was the most crucial. The betweenness values for the remaining nodes changes after the edge with the highest betweenness is removed [46]. Calculating the betweenness only once would result in the edge with a lower betweenness value not being removed until later and thus not separating the communities until later. Re-calculating the betweenness values after the removal of the first edge would raise the value of the originally lower valued edge and thus resulted in the separation of the network into more accurate communities [46].

To test their algorithm, Newman and Girvan first generated networks with a known community structure. They were able to correctly classify more than 90% of the nodes in their trials on the generated graphs of 128 nodes, equally divided into four communities [46]. They then tested their algorithm on the often studied social network, Zachary's karate club [47]. They tested the algorithms using the shortest path and the random-walk betweenness methods and found that the shortest path method perfectly divided the network into its correct communities, while the random-walk betweenness method miss-classified just one node. They also demonstrated that omitting the re-calculation step resulted in not being able to determine the correct communities in Zachary's karate club. To test their algorithm on a larger network, they created a collaboration network from authors and references cited in the bibliography page from their paper. This network had 235 scientists in total, and they showed their algorithm was able to accurately divide the largest component of 140 scientists into 13 communities [46].

### 3.3.3 Fast Community Detection Algorithm

The betweenness community detection algorithm (see Section 3.3.2) was shown to be effective in discovering communities in the tested social networks. One of the drawbacks, however, is the long computation time to compute betweenness measures, especially with large networks. Newman [48] introduced an effective community detection algorithm that is computationally much faster. This fast algorithm is based on the modularity (described in Section 2.2.3) of a network. Since higher modularity theoretically indicates a stronger community structure, the approach is to optimize the modularity value $Q$ of the potential communities. Since optimizing the values over all possible divisions would be extremely expensive (it has an exponential run time and therefore is not feasible for networks with greater than twenty to thirty nodes), the authors employed a standard greedy optimization algorithm [48].

The fast algorithm is an agglomerative method, in contrast with the betweenness algorithm. Each vertex starts as the sole member of one of $n$ communities. Pairs of communities are repeatedly joined based on what would result in the highest modularity ($Q$) value. Since the only pair of communities that will result in an increase in modularity will necessarily contain an edge between them, the algorithm only has to look at $m$ connections, where $m$ is the number of edges. The algorithm keeps track of the modularity as it progresses, so only calculating the change in modularity $\Delta Q$ is required. This can be calculated in constant time. The algorithm was run on the same networks that the *betweenness algorithm* above had previously been tested on and the results were nearly identical. The main difference was the computation time. For example, on a 1275-node jazz musician network, the *fast algorithm* ran to completion in about one second of CPU time, where the *betweenness algorithm* took more than three hours to reach very similar results on identical hardware [48].

# CHAPTER 4: Methodology

Our goal was to develop a method to efficiently and effectively determine which email addresses found on a secondary storage device would be useful to a forensic analyst. Preliminary findings in this area by Green were encouraging; our work builds on these results. Specifically, we analyze Green's graphs of email addresses extracted from secondary storage devices [3]. Green's work showed that groups of email addresses stored in the same region on the device could be further analyzed to discover communities of email addresses with similar properties [3]. This chapter describes the graph creation process based on the methodology defined by Green [3], and then details our methods used in determining which communities could be classified as useful and which graphs could be ignored by an analyst conducting an investigation.

## 4.1 Classification Criteria

A useful email address is one in which the user(s) of the device has communicated with. A 'Useful' group of email addresses is one that contains some number of useful email addresses. 'Not-Useful' groups contain no useful email addresses. The majority of email addresses found in raw storage on a device were not-useful and were found in 'Not-Useful' groups. For example, some groups consisted of addresses all in the form of 'username@microsoft.com' and other groups were in the form of 'some_command@2x.pn.' These email addresses were artifacts of different software or applications and gave no information about the social network of the owner of the device.

## 4.2 The Dataset

Our dataset consisted of 400 graphs, namely 40 graphs from each of the 10 drives created by Green [3]. The 400 graph files were produced from images of 10 different secondary storage devices provided by volunteers. The drives ranged in size and contained a variety of today's most popular operating systems, including Windows, OSX, and Linux.

### 4.2.1 Creating Graph Files from a device

The process to create a graph file from a device is as follows:

1. A image of the volunteer's hard drive was created.
2. A feature file of email addresses found on the image along with their byte-offset was created using bulk_extractor (see Section 2.4.1).
3. The betographer script (see Section 2.4.2) was run on the feature to create a graph file (.gexf).

Figure 4.1 shows the process of creating a graph file from a device.

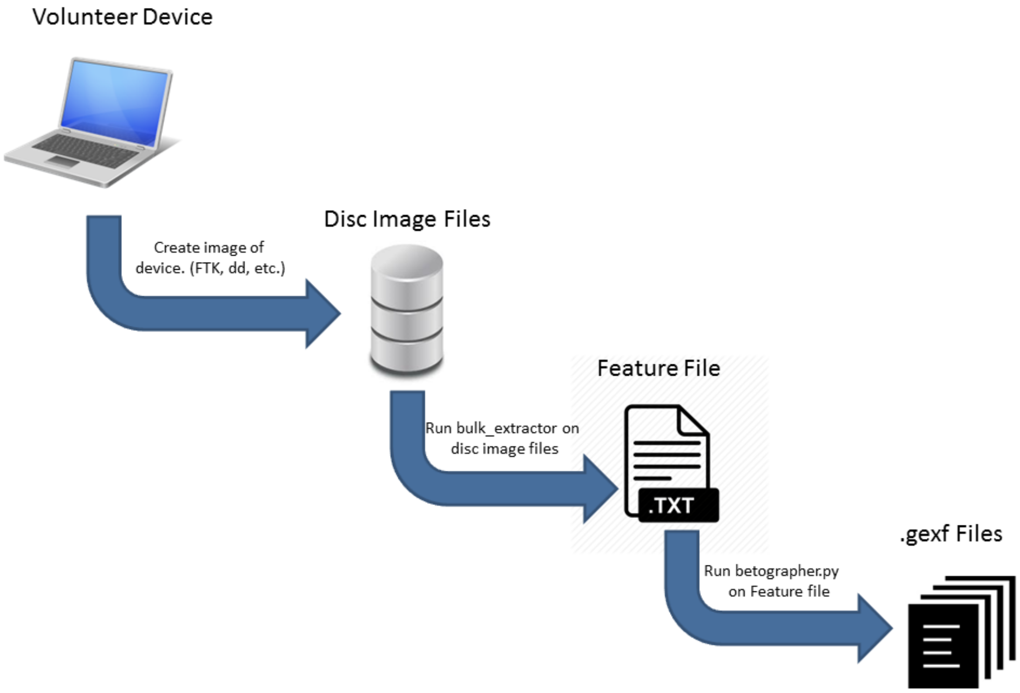

Volunteer Device
Create image of device. (FTK, dd, etc.)
Disc Image Files
Feature File
Run bulk_extractor on disc image files
.TXT
.gexf Files
Run betographer.py on Feature file

Figure 4.1. Process of Creating Graph Files from a Digital Device

### 4.2.2 Graph Structure

Figure 4.2 shows Gephi's representation of a network created from an image's graph file according to the above process. The network has over 25,000 nodes, each one representing a different email address and over 30,000 edges connecting nodes/email addresses. The colors represent different connected components of nodes. This graph has over 11,000 connected components. Over half of the connected components consisted of a single node with no edges indicating that they were located on their own in storage (a single node represents a trivially connected graph). This was generally the case with graphs representing entire images. Green showed that the useful components were generally in the largest 20 components [3]. Thus, betographer was used to create a graph file for each of the top 20 components for each device.

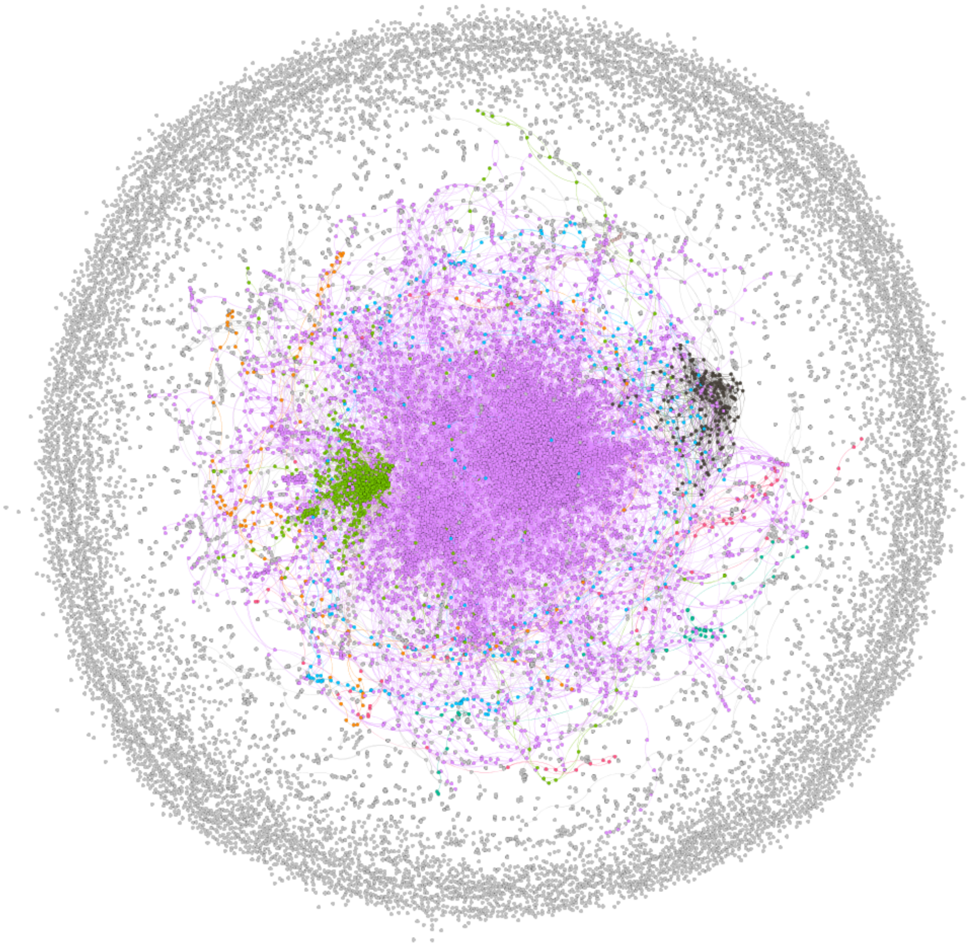


Figure 4.2. Graph of entire image's email addresses and their connections, as shown by Gephi. Different colors represent different connected components of email addresses.

### 4.2.3 Window Sizes

Two different window sizes were used in creating the graphs, 256-bytes and 128-bytes, resulting in 20 graphs for each window size per device. Figures 4.3 and 4.4 show a notional example of how a graph is created using 256-byte and 128-byte windows, respectively. The location of the email addresses on the line is the same on the left side in both figures, but we can see how the graphs differ depending on what the window size is. The graph corresponding to the 256-byte window in figure 4.3 is connected. We also see in that graph the edge between c@email.com and a@email.com is thicker or weighted indicating that c@email.com and a@email.com were co-located within 256 bytes of each other on more than once as seen on the line. The graph created using a 128-byte window shown in Figure 4.4 shows a disconnected graph because c@email.com is not located within 128 bytes of any other email addresses. It does not have any thick or weighted edges, since each pair of email addresses is only co-located together at one location on the line.

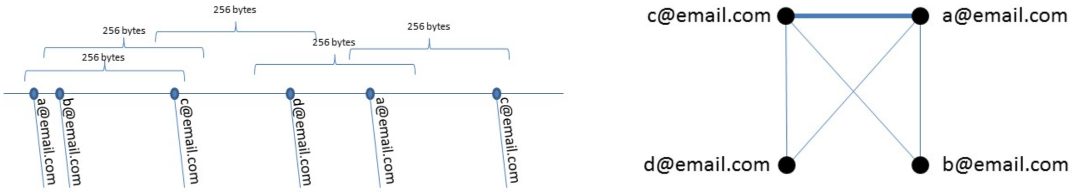


Figure 4.3. 256-byte window example with corresponding graph. Email addresses stored within 256 bytes of each other have an adjacent edge between them represented in the graph on the right. Email addresses that are co-located in multiple location, have a thicker edge between the in the graph.

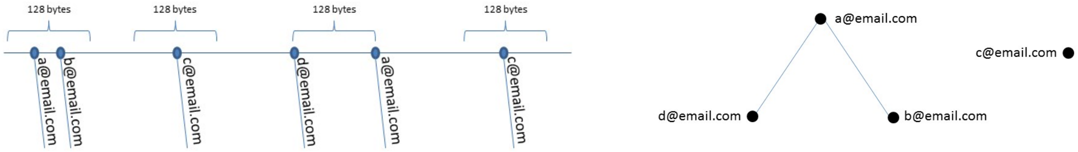


Figure 4.4. 128-byte window example with corresponding graph. Email addresses stored within 128 bytes of each other have an adjacent edge between them represented in the graph on the right. c@email.com is not co-located within 128 bytes of any other email addresses, resulting in a disconnected graph.

## 4.3 Reducing Graphs to Feature Vectors

Out of the sample of 400 networks, we wanted to identify which ones contained useful email addresses. To determine this, we pursued a classification strategy based on global topological features of connected components. This strategy can be viewed as a simplification of the technique proposed by Li et al. [25]. Creating a feature vector involves first calculating the attributes of the graph, and second, normalizing them.

### 4.3.1 Extracting Graph attributes

We used algorithms from NetworkX (see Section 2.4.4) to calculate the majority of each graph's attributes and we wrote a Python script to run these algorithms on each of the .gexf files. We also wrote a Python script to calculate several attributes not currently supported by NetworkX. In total, we calculated 41 different attributes for each graph. The code used to compute the attributes is included in Appendix A.1. The attributes ranged in complexity from basic graph characteristics such as the graph's size, order, and average degree to more complex, computationally expensive algorithms, such as the centralities and degree distributions of a graph. Table 4.1 at the end of the chapter contains a list of all attributes and the methods used to calculate them.

To test the distributions, we created a histogram of the desired feature values for each node in a graph, and used the Python package, power law to determine which distribution the feature was closest to and by how much. The exponent of the fitted curve was also used as an attribute for analysis.

### 4.3.2 Normalization

The majority of the values for the different features ranged between 0 and 1. However, some, such as the size, order, maximal matching, average degree, and average neighbor degree ranged from 0 to the tens of thousands. To prevent over-valuing these attributes, we normalized them, and attempted classification using both the normalized and non-normalized data. To normalize size and order, we divided the size/order, of the connected component by the size/order, of the graph of the entire image. We used range normalization (see Section 2.2.6) for the average degree and average neighbor degree values ($\frac{x-min}{max-min}$). The minimum and maximum values were taken out the top 20 connected components of an image.

## 4.4 Experiments

Each individual test for our experiments was repeated 10 times using the cross validation sampling method using 5 folds, and we took the average over the 10 trials. We designed our experiments to answer the following questions:

1. Is it possible to correctly classify a network as being useful or not, based on the graph's underlying topological structure?
2. Does the size of the window used to create email graphs have an impact on our ability to classify them correctly?
3. What attributes are most effective for classifying the graphs in our dataset?
4. Does our ability to correctly classify our graphs improve when we train against a multi-class labeling scheme, as opposed to a binary scheme in which the only labels are 'Useful' and 'Not-Useful'?

### 4.4.1 Experiment 1: All graphs, All attributes

We first wanted determine if our main hypothesis was viable (Question 1). To test this, we computed attributes for all 400 graphs and labeled each graph as being useful or not based on whether or not the email addresses of a graph were known to the user of the device. We then used Orange (see Section 4.5) to test 4 different supervised learning algorithms (see Section 2.3) on the data. The algorithms were:

1. Classification Tree
2. SVM
3. Logistic Regression
4. Naive Bayes

Out of the 400 graphs, there were 80 graphs that contained fewer than 20 nodes and 40 of these came from one particular device. This device was the only SSD from the sample of drives and we wanted to test the learning algorithms without the SSD as we felt we would not be able to gain much useful information from those graphs. Also, with the smaller graphs, many of the attribute values were 0 or did not provide much useful information about the graph. Therefore, we filtered out the graphs that contained fewer than 20 nodes and re-ran the larger graphs through the same learning algorithms. By removing the graphs with fewer than 20 nodes, our intent was to test whether or not there was a significant difference in the classification results before and after removing smaller graphs.

### 4.4.2 Experiment 2: 128-byte Window vs. 256-byte Window, All Attributes

Our next experiment was designed to determine if there was a difference in the classification results depending on whether the graph files were created using a 128-byte window or a 256-byte window (Question 2). This experiment was identical to Experiment 1, except we divided our dataset into two subsets, one in which the graphs were constructed using a 128-byte window, and one in which graphs were constructed using a 256-byte window. All 41 attributes were used with all four learning algorithms and the tests were run with and without the smaller graphs.

### 4.4.3 Experiment 3: Selected Attributes

In Experiments 1 and 2, we used all 41 different attributes for each graph in testing the learning algorithms. In this experiment, we tested the learning algorithms using select groups of attributes to see how they performed (Question 3). We started by testing with just four basic attributes that could be calculated using only the graph's size and order to see how well the learning algorithms performed. The four basic attributes were:

- Normalized Order: number of nodes in the component divided by the number of nodes in entire image.
- Normalized Size: number of edges in the component divided by the number of edges in entire image.
- Average degree.
- Density.

Each graph's size and order for an image could be found in the summary file created by betographer, thus removing the need to analyze and extract attributes from each graph. We then added just one attribute, the average neighbor degree (r-normalized) to the four basic attributes and tested the learning algorithms. Next, we selected what appeared to be the top

10 attributes for indicating a graph's usefulness using several techniques. We looked at the scatter plots for each attribute plotted against their class labels ('Useful' or 'Not-Useful'). In these plots, we were looking for tight clusters of attribute values for 'Useful' networks and attributes for which the range of values for 'Useful' networks was different from the range for 'Not-Useful' networks.

The scatter plot in Figure 4.5 shows the average neighbor degree (r-normalized) by class. It can be seen that the majority of 'Useful' networks had a higher value than the 'Not-Useful' networks. There were two 'Useful' networks that had slightly lower values but those values were still noticeably higher than the majority of the 'Not-Useful' Graphs. The average neighbor degree (r-normalized) was selected as one of the better indicators, in contrast to modularity. Figure 4.6 shows modularity is a poor indicator: the modularity values for 'Useful' networks were generally spread out and the majority of the 'Not-Useful' networks' modularity values fell within the same ranges as the 'Useful' networks.

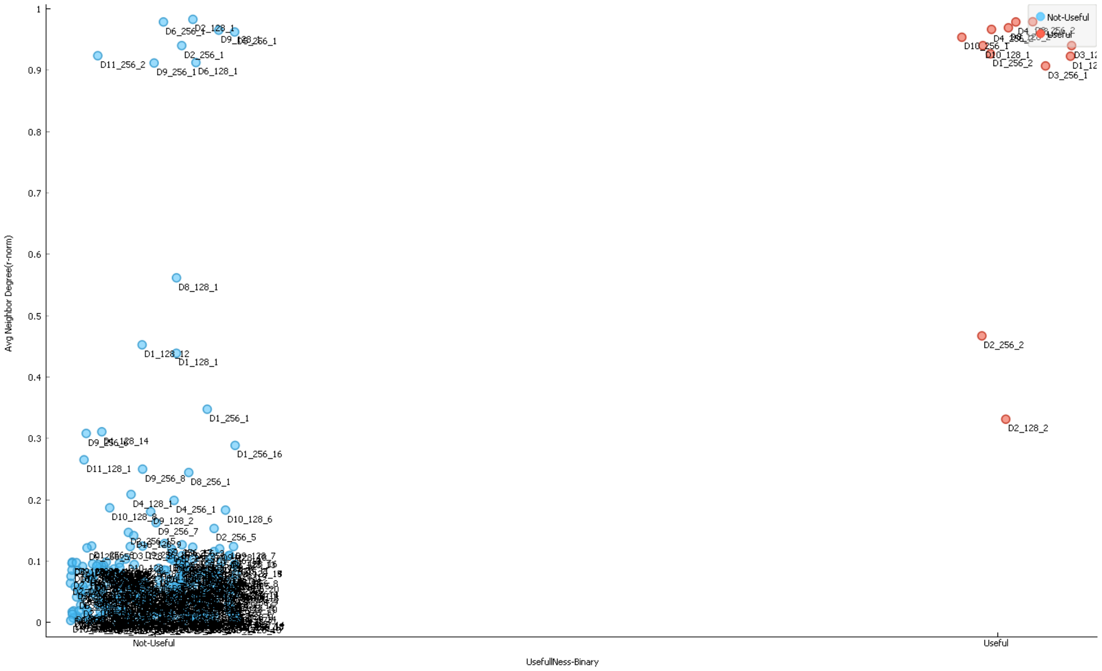


Figure 4.5. Good indicator example: scatter plot of average neighbor degree (r-normalized) vs. Usefulness.

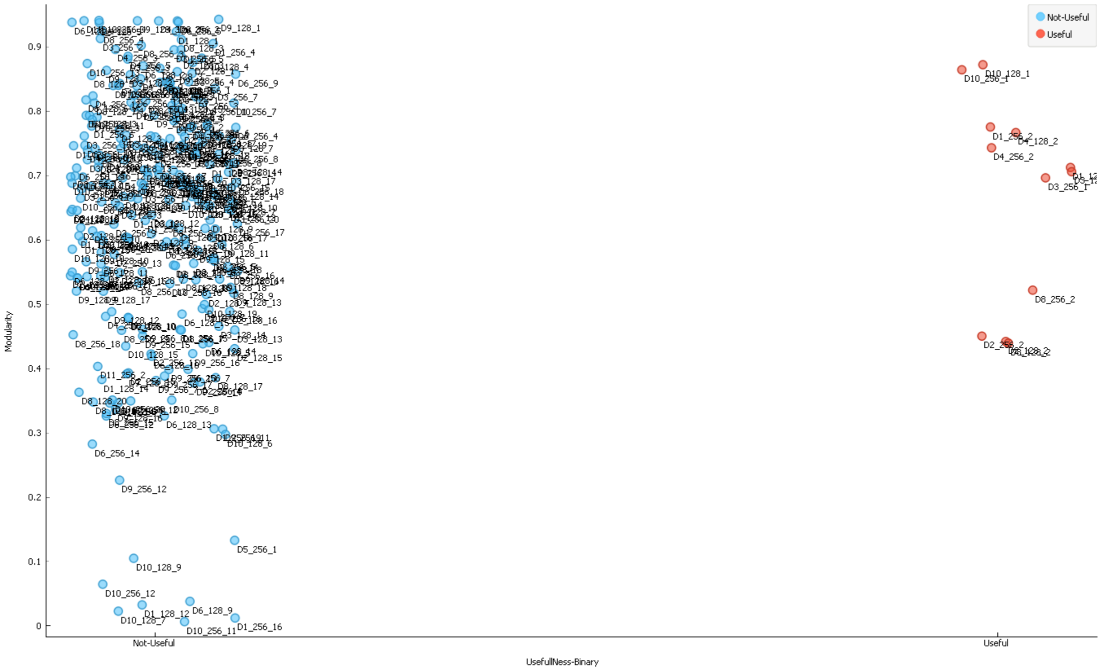


Figure 4.6. Poor indicator example: scatter plot of modularity vs. Usefulness.

The scatter plots offered a quick look at the clustering of different attribute values. We also looked at the box plots for each attribute separated into their usefulness classes which provided the mean, median, quartiles, and standard deviation of the attribute values for members of each class. Figures 4.7 and 4.8 show the box plots for the two usefulness classes for transitivity and the average clustering coefficient, respectively. It can be seen that the majority of the transitivity values were much lower in the 'Useful' networks than the transitivity values for the average clustering coefficient showing that transitivity should be a good indicator. The average clustering coefficient appears to be a poor indicator based on its box plots shown in Figure 4.8.

We also looked at classification trees produced by Orange to see what attributes were chosen. Figure 4.9 shows a decision strategy using Orange's regression tree viewer widget. At the top of the tree, it first looks at the proportion of nodes compared to the entire image. Smaller networks under .005 are immediately classified as 'Not-Useful'. The next test is degree centrality where values less than 0.125 are classified as being 'Not-Useful.' Those networks not classified yet and greater than 0.125 then go to the betweenness centrality test. This process continues until a decision is reached at a leaf node as seen in Figure 4.9.

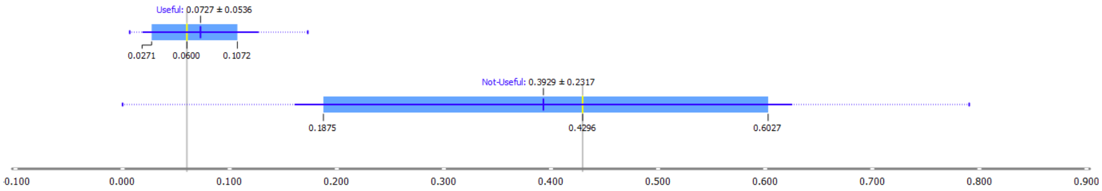


Figure 4.7. Box plots showing transitivity values with respect to usefulness class. The relatively small area of overlap marks this attribute as a strong indicator.

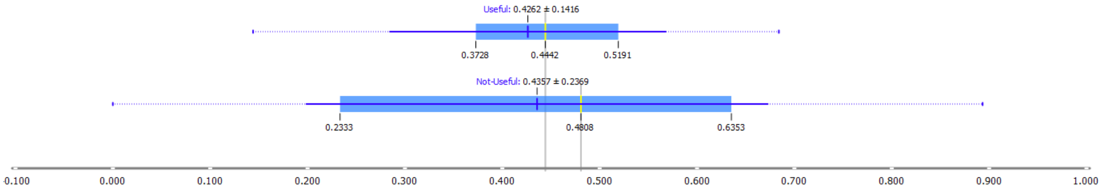


Figure 4.8. Box plots showing average clustering coefficient values with respect to the usefulness class.

### 10 Best Attributes

From our techniques mentioned in Section 4.4.3, we found that our 10 best attributes were (in no particular order):

- Density
- Average neighbor degree (r-normalized)
- Pearson coefficient
- Transitivity
- Highest Betweenness centrality
- Maximal matching (divided by number of edges)
- Maximal matching
- Number of nodes (percentage from entire image)
- Degree distribution (best fit)
- Degree Distribution value

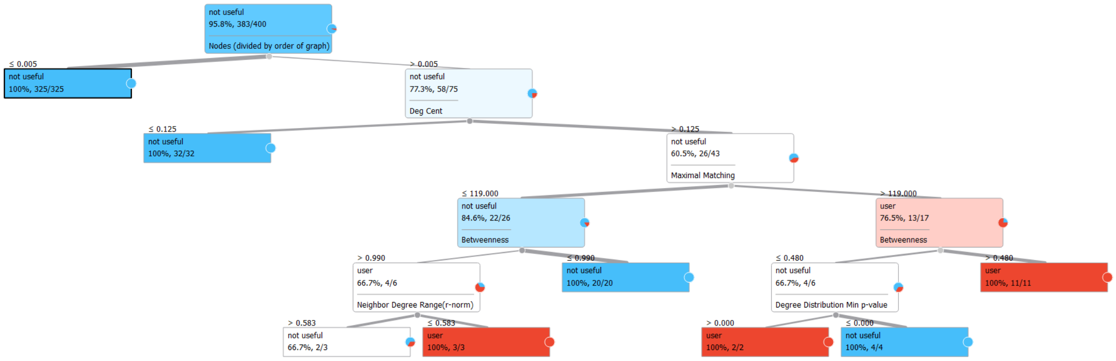


Figure 4.9. Classification tree for graphs with more than 20 nodes.

### 4.4.4 Experiment 4: Multiple Classes

Manual inspection of the networks revealed that connected components tended to contain email addresses that were similar in terms of their origin or function. This observation was not unexpected, since these components were formed from addresses residing near each other in storage, and were thus likely created by the same applications. However, while 'Useful' networks were a relatively small, homogenous group, 'Not-Useful' networks often fell into a variety of clearly recognizable categories. For this reason, we performed an experiment to determine whether subdividing the 'Not-Useful' networks class into 8 subclasses corresponding to our empirical observations would allow us to train better classifiers. The 9 classes with the 'Not-Useful' divided into 8 subclasses and 'Useful' class changed to 'Owner' class were:

- **Owner:** email addresses that the owner communicated with (my_best_friend@yahoo.com)
- **Database:** email addresses in the form of SOME_NAME@database.com
- **Ubuntu:** email addresses in the form of username@ubuntu.com or username@debian.org
- **Microsoft:** email addresses in the form of username@microsoft.com
- **Certificates:** email addresses in the form of LONG_RANDOM_STRING @certificate_site.com (5E6119578EEEC642A96EC666DC7940C03C3D15BE @UPD-CHU15.easf.csd.organiziation.mil)
- **Broadcast:** email addresses in the form of some_group@some_school.edu
- **Username:** email addresses in the form of owner's username@some_website.com (username@google.com)
- **Mac Artifact:** email addresses appearing to be MAC commands. some_command@2x.pn

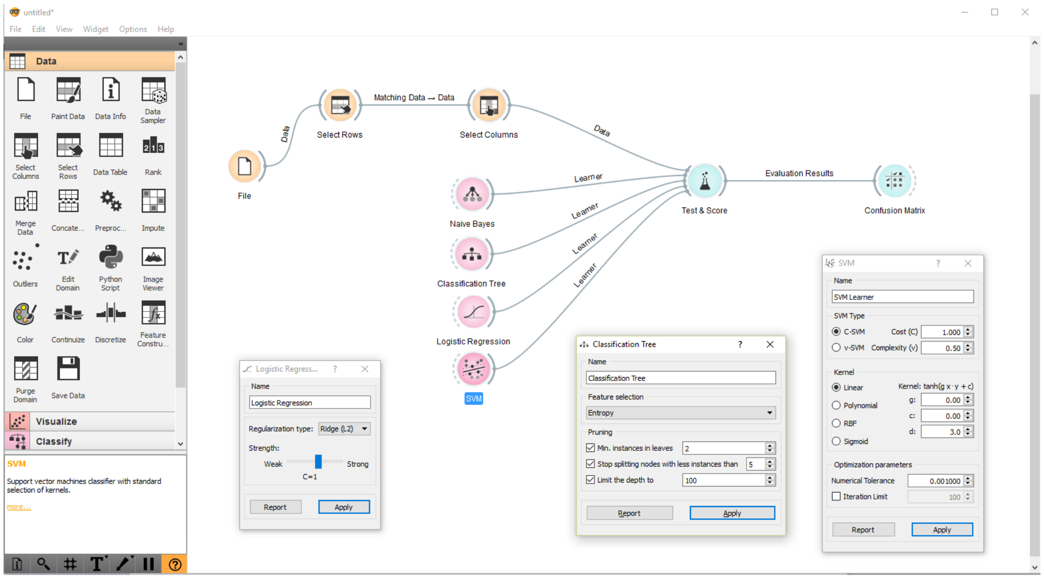
- **Other:** email addresses that we could not fit to one of the above classes

We found that by creating additional classes, the percentage of ‘Not-Useful’ networks (now ‘Other’) decreased from 95% of the data set to roughly 50%. We tested the learning algorithms using the multiple classes and compared the results to our experiments run with two classes.

## 4.5 Orange Setup

Figure 4.10 shows a screenshot of our Orange template along with the settings and parameters used when testing the learning algorithms. The file widget in the upper left of the figure represents the starting point, where our graph attribute data is fed into the test framework. The Select Rows widget filters to restrict graph files to those with greater than 20 nodes. In the Select Columns widget, we selected which attributes to use and selected the target variable (usefulness) we wanted to run the classification on. The selected data was then fed into the Test & Score widget.

The Test & Score widget evaluated the four learning algorithms. The parameters used for logistic regression, classification tree, and SVM are shown in the windows in Figure 4.10. We ran each experiment using 5-fold validation. The majority (96%) of our networks were ‘Not-Useful’ so we used stratified sampling (see Section 2.4.5). We used a Confusion Matrix to summarize our results.



Logistic Regress...

Name: Logistic Regression

Regularization type: Ridge (L2)

Strength: Weak – Strong, C=1

Report | Apply

Classification Tree

Name: Classification Tree

Feature selection: Entropy

Pruning

☑ Min. instances in leaves: 2

☑ Stop splitting nodes with less instances than: 5

☑ Limit the depth to: 100

Report | Apply

SVM

Name: SVM Learner

SVM Type

● C-SVM, Cost (C): 1.000

○ v-SVM, Complexity (v): 0.50

Kernel

● Linear

○ Polynomial

○ RBF

○ Sigmoid

Kernel: tanh(g x·y + c)

g: 0.00

c: 0.00

d: 3.0

Optimization parameters

Numerical Tolerance: 0.001000

□ Iteration Limit: 100

Report | Apply

Figure 4.10. Screenshot of Orange template used for classification with the parameters that were used and settings used.

| Attribute | Attribute Source |
|---|---|
| Number of Nodes | Summary file created by Betographer |
| Number of Edges | Summary file created by Betographer |
| Number of Nodes (divided by order of image) | Summary file created by Betographer |
| Number of Edges (divided by size of image) | Summary file created by Betographer |
| Average Degree | Summary file created by Betographer |
| Average Degree (r-normalized) | Summary file created by Betographer |
| Density | Summary file created by Betographer |
| Average Neighbor Degree | NetworkX |
| Minimum | Python Script |
| Maximum | Python Script |
| Range (max - min) | Python Script |
| Average Neighbor Degree (r-normalized) | Python Script |
| Minimum (r-normalized) | Python Script |
| Maximum (r-normalized) | Python Script |
| Range (r-normalized) | Python Script |
| Modularity | Community module |
| Average Clustering Coefficient | NetworkX |
| Pearson Coefficient | NetworkX |
| Maximal Matching (Size) | NetworkX |
| Maximal Matching(Size)/Number of Edges in Graph | Python Script |
| Highest Degree Centrality | NetworkX |
| Highest Closeness Centrality | NetworkX |
| Highest Betweenness Centrality | NetworkX |
| Rich Coefficient (Maximum) | NetworkX |
| Rich Coefficient (Minimum) | NetworkX |
| Average Shortest Path Length | NetworkX |
| Maximum Core | NetworkX |
| Minimum Core | NetworkX |
| Order of Maximum Core | NetworkX |
| Order of Minimum Core | NetworkX |
| Transitivity (Global Clustering Coefficient) | NetworkX |
| Diameter | NetworkX |
| Radius | NetworkX |
| Range (Diameter - Range) | Python Script |
| Order of Center | NetworkX |
| Order of Periphery | NetworkX |
| Best Degree Distribution Fit | Power Law Module |
| P-value for Best Degree Distribution | Power Law Module |
| Degree Distribution Exponent Value | Power Law Module |
| Size of largest Clique | NetworkX |

Table 4.1. All 41 Attributes with source of Algorithm.

# CHAPTER 5: Results and Analysis

This chapter details the results and provides analysis of our four experiments. The results of our initial experiment—in which we tested four different learning algorithms on every attribute for every graph using the classes 'Useful' or 'Not-Useful' served as our benchmark to compare with and to improve upon. After presenting these results, we examined the performance of the learning algorithms when looking at different groups of the networks (Experiment 2) and using different subsets of attributes (Experiment 3). Finally, we show that training against a broader range of classes improved our outcome considerably (Experiment 4).

We used the area under curve (AUC), classification accuracy (CA), F1, precision, and recall scores (see Section 2.3.6) as our performance measures. We tended to focus on the AUC and F1 scores as the more important performance measures. The AUC (see Section 2.3.6) takes into account more than just the classification accuracy by looking at both the true positive and false positive rates and has been shown to be a better performance measure when testing learning algorithms [49]. Because over 90% of our networks belonged to one class, the fact that the majority of our learning algorithms achieved a classification accuracy over 90% did not reveal any insight regarding their effectiveness. We focused on the F1 score because it factors in both the precision and recall rates (F1 scores closer to 1 indicated a better performing learning algorithm).

## 5.1 Experiment 1: All Graphs, All Attributes

Table 5.1 shows the results of our first experiment, in which we used all 400 networks with all 41 attributes to test how well the four learning algorithms performed in predicting whether or not a network was 'Useful' or 'Not-Useful'.

The values in Table 5.1 help provide an answer to our initial question from Section 4.4: it is indeed possible to classify a network as being useful or not based on its underlying topological structure. Although not perfect, the initial results demonstrate the potential of this method. The recall rates of .875 and .750 for Naïve Bayes and SVM, respectively,

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.905 | 0.932 | 0.509 | 0.359 | 0.875 |
| Classification Tree | 0.709 | 0.960 | 0.466 | 0.501 | 0.438 |
| Logistic Regression | 0.776 | 0.973 | 0.600 | 0.621 | 0.562 |
| SVM | 0.868 | 0.978 | 0.727 | 0.706 | 0.750 |

Table 5.1. Evaluation results using all 400 graphs with all 41 attributes for 2 classes: Useful vs. Not-Useful.

indicate that they correctly captured 87.5% and 75.0% of the 'Useful' networks, which is promising because the 'Useful' networks only made up roughly 4% data. Naïve Bayes had a lower precision rate at .359 indicating that it tended to over-classify networks as being useful when they were not; this was generally the case throughout the rest of the experiments.

Naïve Bayes and SVM also perform well when looking at their AUC, with areas at .905 and .868, respectively. It is generally understood that areas greater than .90 could be interpreted as being excellent, areas between .80 and .90 as good, and areas in between .70 and .80 as average [32]. Classification Tree and Logistic Regression did not perform particularly well in our first experiment with every graph, but they did capture roughly half of the useful networks.

### 5.1.1 Removing Smaller Graphs

Table 5.2 shows the evaluation results for runs against graphs with more than 20 nodes. As expected, the learning algorithms performed better, with their scores improving in virtually every performance measure. This was due to the fact every 'Useful' network except for the 'Useful' networks from the SSD had thousands of nodes. The larger networks were more complex and this complexity was captured in their attribute values which were not captured in the smaller networks. For example, in smaller graphs, the highest centrality value for every graph was mostly at 1, which was not the case for the larger networks. Also, the smaller graphs were not able to be fitted to any degree distributions.

Table 5.2 shows the AUC for all four learning algorithms on average increased .045. Naïve Bayes' precision remained low but its recall rose to 1 indicating it did not miss any 'Useful' networks. Lower precision rates indicated that the learning algorithm predicted more networks as being 'Useful' when they were 'Not-Useful'. Details can be seen in the Naïve

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.961 | 0.926 | 0.511 | 0.343 | 1.000 |
| Classification Tree | 0.741 | 0.963 | 0.512 | 0.528 | 0.500 |
| Logistic Regression | 0.825 | 0.971 | 0.640 | 0.615 | 0.667 |
| SVM | 0.912 | 0.984 | 0.800 | 0.769 | 0.833 |

Table 5.2. Evaluation results on graphs with 20 nodes or greater using all 41 attributes for 2 classes: Useful vs. Not-Useful.

Bayes confusion matrix in Table 5.3, which shows that Naïve Bayes predicted 35 graphs as being ‘Useful’ when only 12 of those were actually ‘Useful’. This would give the analyst 23 ‘Not-Useful’ networks to analyze and manually discard, which is not ideal. Nonetheless, our results demonstrated a clear advantage to discarding small graphs. The rest of our experiments were conducted using graphs with more than 20 nodes and the evaluation results in Table 5.2 serve as the new benchmark to improve upon.

| | | *Predicted* | | |
|---|---|---|---|---|
| | | Not-Useful | Useful | ∑ |
| *Actual* | Not-Useful | 275 | 23 | **298** |
| | Useful | 0 | 12 | **12** |
| | ∑ | **275** | **35** | **310** |

Table 5.3. Confusion Matrix for Naïve Bayes on graphs with 20 nodes or greater using all 41 attributes for 2 classes: Useful vs. Not-Useful.

## 5.2 Experiment 2: 128-Byte Window vs. 256-Byte Window, All Attributes

Our initial experiment demonstrated that using topological features from a graph created from email addresses and their proximity to each other on a storage device could be used to classify those graphs as useful or not. Our second experiment demonstrates that these results hold for more than one choice of window size. We find that both visual inspection of the networks and comparison of classification results corroborate this conclusion.

In the visual inspection phase of this experiment, we first visually inspected the email addresses of each network for both window sizes for each device. We found that the 20 largest networks for the two different window sizes appeared to match up for the majority of the networks for each device. For example, the largest network created using the 128-byte window on several drives consisted of email addresses of developers associated with Ubuntu. The same email addresses showed up in the network of the same drives when using a 256-byte window. This was generally the case for all components throughout the data set. There were a few instances where the ordering of the networks were different between the two window sizes of the same device. For example, a group of Microsoft email addresses showed up in network 16 when using a 128-byte window on a particular drive, and the same group of email addresses showed up in network 17 for the same drive when using 256-byte window. However, regardless of what window size was used, it appeared that the 20 largest networks were similar in the email addresses that they contained.

The two different window sizes did produce different graphs—that is, although they contained similar sets of email addresses, their topology showed considerable variation, and naturally their attribute values were also different. For the 'Useful' graphs, the number of nodes created using the 128-byte window was on average 77% the number of nodes created when using a 256-byte window. The decrease in the number of edges was even more pronounced with the number of edges in a 128-byte window averaging only 49% of the number of edges created using a 256-byte window. Of particular concern was that capturing 77% of nodes when using a smaller window would leave out relevant email addresses when trying to reconstruct the social network of the user. However, we found that the 'Useful' graphs were generally the top 1 or 2 largest components for each graph and contained thousands of email addresses. Through closer inspection and analysis, we found that the relevant email addresses were captured when looking at the cores of the graphs regardless of the window size.

Table 5.4 shows the performance results of the learning algorithms on the networks created using both window sizes, with 128-byte results on the left side and 256-byte results on the right. From the results, each method performs roughly the same throughout the different measures.

The results in Table 5.4 show that the classification tree method performed slightly better when a 128-byte window was used. SVM also performed slightly better when using a 128-byte window. Most interesting was how well Logistic Regression performed. The AUC for the 256-byte window was at .997 which appeared high. We ran the test using the 256-byte

| **AUC** | 128-Byte | 256-Byte | Difference |
|---|---|---|---|
| **Naïve Bayes** | 0.956 | 0.957 | -0.001 |
| **Classification Tree** | 0.743 | 0.688 | 0.055 |
| **Logistic Regression** | 0.913 | 0.997 | -0.084 |
| **SVM** | 0.907 | 0.827 | 0.080 |
| **CA** | | | |
| **Naïve Bayes** | 0.922 | 0.917 | 0.005 |
| **Classification Tree** | 0.968 | 0.960 | 0.008 |
| **Logistic Regression** | 0.987 | 0.994 | -0.007 |
| **SVM** | 0.974 | 0.974 | 0.000 |
| **F1** | | | |
| **Naïve Bayes** | 0.500 | 0.480 | 0.020 |
| **Classification Tree** | 0.545 | 0.424 | 0.121 |
| **Logistic Regression** | 0.833 | 0.923 | -0.090 |
| **SVM** | 0.714 | 0.667 | 0.047 |
| **Precision** | | | |
| **Naïve Bayes** | 0.330 | 0.316 | 0.014 |
| **Classification Tree** | 0.600 | 0.467 | 0.133 |
| **Logistic Regression** | 0.833 | 0.857 | -0.024 |
| **SVM** | 0.625 | 0.667 | -0.042 |
| **Recall** | | | |
| **Naïve Bayes** | 1.000 | 1.000 | 0.000 |
| **Classification Tree** | 0.500 | 0.294 | 0.206 |
| **Logistic Regression** | 0.833 | 1.000 | -0.167 |
| **SVM** | 0.833 | 0.667 | 0.166 |

Table 5.4. Evaluation results for learning algorithms on graphs created using a 128-byte window and 256-byte window. Difference is shown with 128-byte column as reference point for difference.

window several times, each time achieving the same results. Table 5.5 shows the confusion matrices for both window sizes when run with Logistic Regression. The difference is in the false positives in the upper right. Using a 128-byte window size resulted in 11 false positives where using a 256-byte window only resulted in 1 false positive. We were not able to find a specific reason on why Logistic Regression performed so well using a 256-byte window; we leave that for future work.

One benefit of using a 128-byte window would be fewer email addresses for an analyst to look through given a 'Useful' network. However, we concluded that either window size

| **128-Byte Window** | | *Predicted* | | |
|---|---|---|---|---|
| | | Not-Useful | Useful | ∑ |
| *Actual* | Not-Useful | 137 | 11 | **148** |
| | Useful | 0 | 6 | **6** |
| | ∑ | **137** | **17** | **154** |

| **256-Byte Window** | | *Predicted* | | |
|---|---|---|---|---|
| | | Not-Useful | Useful | ∑ |
| *Actual* | Not-Useful | 149 | 1 | **150** |
| | Useful | 0 | 6 | **6** |
| | ∑ | **149** | **7** | **156** |

Table 5.5. Confusion matrices for Logistic Regression test using 128-byte window (left) and 256-byte window (right).

would work just as well when using our method. This was encouraging as it showed that our methodology was not sensitive to small perturbations in parameters such as the window size.

## 5.3 Experiment 3: Select Sets of Attributes

The outcomes of Experiments 1 and 2 affirmed our basic approach, but did not provide insight as to which of our features contributed most to our classification decisions. Experiment 3 showed that certain attributes were better than others for indicating the usefulness of a network.

### 5.3.1 Testing Basic Attributes

Table 5.6 shows the evaluation results when using the 4 basic attributes we selected for our minimal set. Somewhat surprisingly, both Naïve Bayes and Classification Tree performed well. Two major factors contributed to this— 1) The fact that graph size eliminates most bad graphs immediately and 2) Both Naïve Bayes and Classification Tree classify with a very low confidence threshold. Naïve Bayes' AUC was high at .968 but had a lower F1 score at .358. Of particular interest was that Naïve Bayes had a recall rate of 1 and a precision rate at .218 indicating its tendency to over-classify.

This result suggests that using Naïve Bayes with these four basic attributes could be valuable when conducting an initial triage in an investigation on a large number of devices. The attributes are computed with only the size and order of the networks. These values are recorded in a summary file that is created when betographer is run on the feature files, thus eliminating the need to look at individual networks. Having a recall rate of 1 means that nothing would be missed, although a lot of extra networks would be classified as being 'Useful.' However, this could potentially be a good first filter to lower the number of networks to analyze further—especially as a method for intelligently down sampling before training more sophisticated classifiers.

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.968 | 0.861 | 0.358 | 0.218 | 1.000 |
| Classification Tree | 0.867 | 0.974 | 0.692 | 0.643 | 0.750 |
| Logistic Regression | 0.500 | 0.961 | 0.000 | 0.000 | 0.000 |
| SVM | 0.500 | 0.961 | 0.000 | 0.000 | 0.000 |

Table 5.6. Evaluation results for learning algorithms using basic graph attributes: Normalized Order, Normalized Size, Average Degree and Density.

Classification Tree also performed well when tested with the four basic attributes. Its AUC was high (.867) and its F1 score was .692 which compared favorably to Naïve Bayes at .358. Although classification tree might miss more (25%) of the 'Useful' networks, it would further reduce the number of networks for an analyst to look at. Logistic Regression and SVM failed to predict any networks as being 'Useful' and would therefore not be any value when using only the basic attributes.

When tests were rerun with the average neighbor degree (r-normalized) added to the basic four, we saw an immediate improvement with SVM. Table 5.6 shows that SVM's AUC was at .500 using 4 features, indicating that it performed no better than random classification (or just rejected everything). With the addition of just the one attribute, we saw SVM's AUC increase to .830 with a F1 score at .583 as shown in Table 5.7. Naïve Bayes and Classification Tree showed improvements as well with the addition of the average neighbor degree attribute. However, Logistic Regression still failed to perform better than random classification.

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.968 | 0.939 | 0.558 | 0.387 | 1.000 |
| Classification Tree | 0.872 | 0.984 | 0.783 | 0.818 | 0.750 |
| Logistic Regression | 0.500 | 0.961 | 0.000 | 0.000 | 0.000 |
| SVM | 0.830 | 0.981 | 0.727 | 0.800 | 0.667 |

Table 5.7. Evaluation results for learning algorithms using basic graph attributes plus the average neighbor degree.

### 5.3.2 Analysis of Attributes

We will now provide a review of the attributes and the roles they played in the different networks.

Figure 5.1 shows an example of a ‘Useful’ network when displayed in Gephi (see Section 2.4.3). We found that all of the ‘Useful’ graphs had this general appearance when using the Force Atlas 2 visualization algorithm. The drive owner’s email address(es) were always found in the center of the graph and the majority of nodes were clustered in the center as seen in the figure. Each user graph had several branches extended from the center cluster of nodes. Each of these branches contained nodes representing email addresses, generally from the same email thread. For example, some branches were made up of email addresses of students in the same program at the same university. Some branches represented email addresses of family members, and some contained email addresses pertaining to a club or athletic teams that the user was part of.

**Size and Order**

We found the ‘Useful’ networks generally had the most nodes and edges compared to the rest of the networks on a device. Depending on the device, ‘Useful’ graphs generally had between a 1,000 and 10,000 nodes with roughly 2 to 3 times as many edges as nodes. The majority of the rest of the networks were generally less than 500 nodes with less than 1,000 edges. Thus, the size and order were good indicators of a graph’s usefulness and were even better normalized.

**Density**

By looking at the networks’ densities (see Section 2.2.1) in scatter and box plots, we determined that it was a good indicator of a network’s usefulness. ‘Useful’ networks tended to be sparse with values less than .05. The sparsity of the user networks made sense in that we found that the user of a device generally had several different groups that they associated with that created different branches and communities originating from the user. The individual email addresses in these branches were located close to each other in storage on the device and were connected to each other in the graph, but the branches were not co-located and thus only connected by going back to the center of the graph. This left the majority of the nodes with a low degree, resulting in an overall low density for the graph.

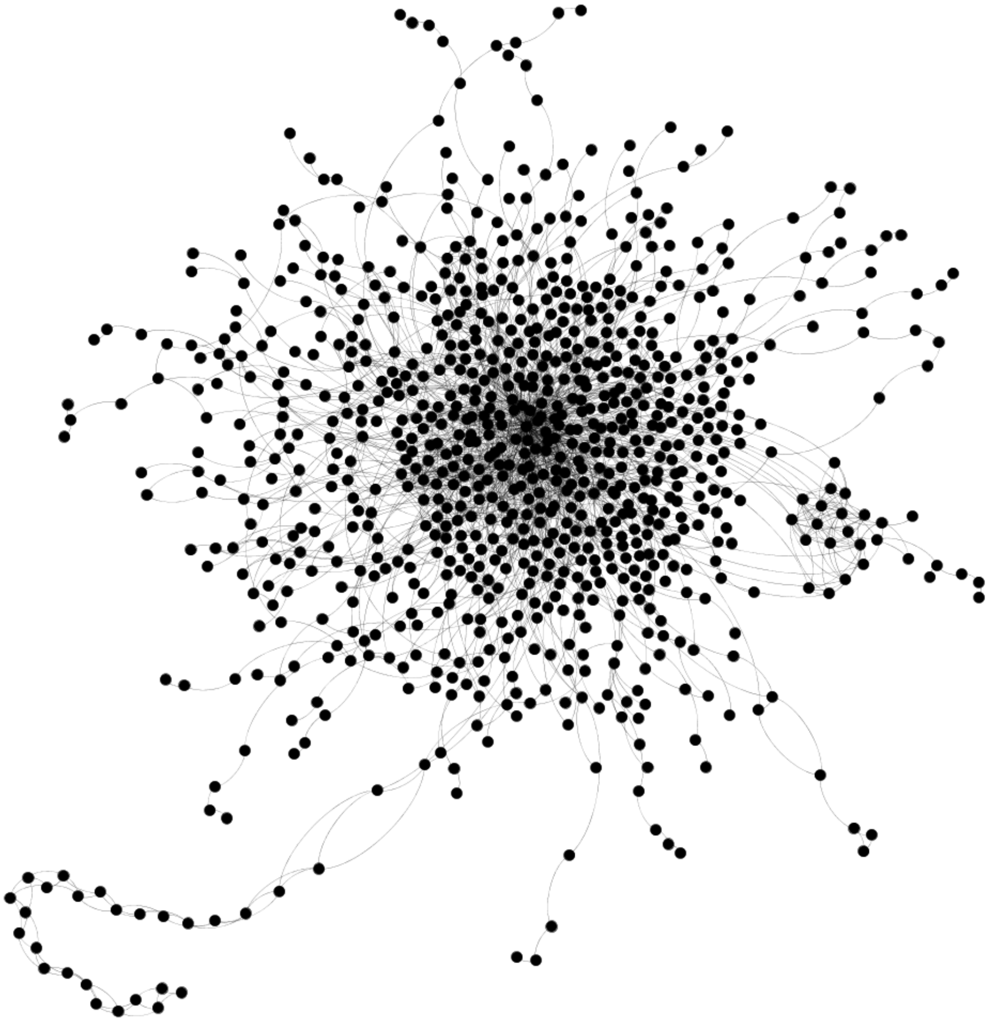


Figure 5.1. Useful network visualized using Force Atlas 2 layout from Gephi.

**Average Neighbor Degree and Degree Distribution**

We found that the average neighbor degree of a network, specifically when r-normalized, was a good indicator for classification. We observed that the 'Useful' networks generally had a few central hubs with a degree that kept the rest of the nodes connected through those central nodes. These hubs were generally the user's email addresses. Since the majority of nodes were connected to a hub with a higher degree, it makes sense that average neighbor degree in 'Useful' graphs would be higher. This phenomenon was also captured when looking at the degree distributions. A few nodes had high degrees and the majority of the rest had small degrees. The Degree Distribution of ‘Useful’ networks all closely matched an exponential distribution and had a higher Degree Distribution value.

### Centralities

We hypothesized that the email addresses with the highest centrality (see Section 2.2.5) values in the ‘Useful’ networks would be the user’s email addresses. This was found to be the case. We looked at the node with the highest centrality and used that nodes’ centrality value as an attribute for the network. This was generally the centrality of the owner’s most often used email address. Out of the three centrality measures we examined (degree, closeness and betweenness), we found that the highest betweenness centrality value offered the best indication of usefulness. Betweenness centrality for a given node is a measure of how often that node acts as a “middle-man” in the shortest path between pairs of nodes in the network. It follows that more often than not, a node would have to go through the owner’s email address to reach another node in the shortest distance.

### Pearson Coefficient

Figure 5.2 shows another graph with low density. If it were classified based on density alone, it would be classified as ‘Useful’. This network, however, was made up of email addresses affiliated with a software program on the device and was ‘Not-Useful.’ It had a Pearson coefficient of .84, which was very high compared to the Pearson coefficients we found in our ‘Useful’ graphs which were between -0.2 and 0.2. The ‘Not-Useful’ network in Figure 5.2 is shown using the same Force Atlas 2 visualization algorithm as in Figure 5.1 and it appears to have a much more organized structure. This structure was seen more often with graphs with higher Pearson coefficients and none were ‘Useful’. We see that most nodes have roughly the same degree and that there are not any nodes that stand out as being more important than the others, contrary to what was seen in the ‘Useful’ networks.

### Maximal Matching

We looked at the maximal matchings (see Section 2.2.2) for each network. More specifically, we used the number of edges in the maximal matching of a network as an attribute. We found the size (number of edges) of the maximal matching in the ‘Useful’ networks were generally larger when compared to the ‘Not-Useful’ networks. However, we attribute this

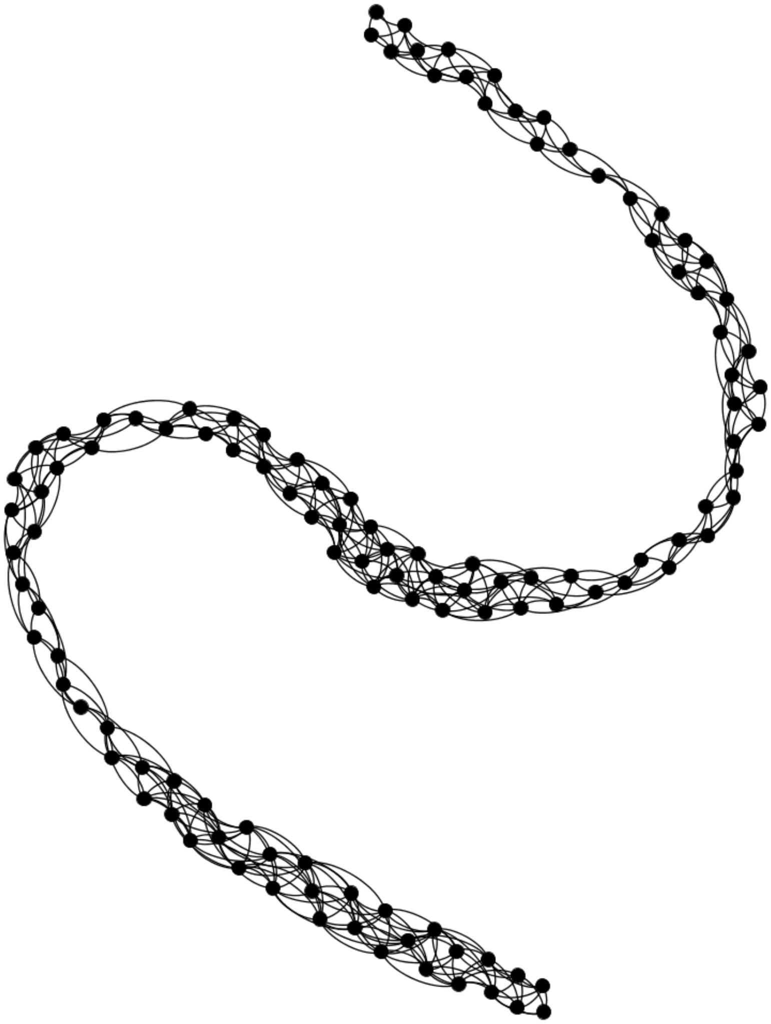


Figure 5.2. Not-useful graph visualized using Force Atlas 2 layout from Gephi with Pearson coefficient of .84.

to the fact that the 'Useful' networks were the larger networks to begin with; it follows that they would have larger maximal matchings. In contrast, we found that the ratio of the size of the maximal matching to the total size of the network was smaller for the 'Useful' than the 'Not-Useful' networks. This is partly due to the 'Useful' networks being less dense. As a reminder, a matching is made up edges that do not share a common vertex. Less dense graphs will have fewer opportunities for their edges to connect to non-adjacent edges.

**Clustering Coefficient**

We looked at two different clustering coefficients (see Section 2.2.3) and found that the global clustering coefficient was a much better indicator than the average clustering coefficient. The global clustering coefficient or transitivity of 'Useful' networks, which measures the ratio of actual triangles to the number possible triangles, was generally lower when compared to the 'Not-Useful' networks. The average clustering coefficient, which calculates the local clustering coefficient for each node and takes the average of all nodes for the entire network had values that ranged between 0.1 and 0.7 for 'Useful' networks and were not easily distinguishable from the 'Not-Useful' graphs.

**Modularity**

Modularity did not appear to be a good indicator in determining whether a graph was associated with a user's social network. We found that the majority of the 'Useful' graphs had a relatively high modularity between 0.5 and 0.8 showing good community structure. This made sense as 'Useful' networks usually displayed distinct groups that formed communities of their own, reflective of a higher modularity value. However, we found that the majority of networks, regardless of their usefulness had the same modularity values (as seen in Figure 4.6).

### 5.3.3 Testing the Top 10 Attributes

Table 5.8 shows the performance results for the learning algorithms using the 10 best attributes from Section 4.4.3. Naïve Bayes showed slight improvements over the same performance measures when tested using all 41 attributes. Logistic Regression performed poorly when run with fewer attributes in our tests. SVM showed the greatest improvements increasing its AUC by more than .004 to .955 and its F1 score increasing by .008 to .880 when run on our 10 best selected attributes. We tested different combinations of attributes outside of our top 10, mixed and matched different combinations, and found that the results in Table 5.8 were generally the best. We did find that Logistic Regression scores increased as we added more attributes.

We saw overall improvements when going from 41 to 10 attributes when testing the learning algorithms. Naïve Bayes still showed a tendency to over-classify and SVM continued to perform well. Logistic Regression performance improved as we added more attributes.

| Method | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.966 | 0.935 | 0.545 | 0.375 | 1.000 |
| Classification Tree | 0.767 | 0.967 | 0.578 | 0.627 | 0.550 |
| Logistic Regression | 0.580 | 0.961 | 0.250 | 0.500 | 0.167 |
| SVM | 0.955 | 0.990 | 0.880 | 0.846 | 0.917 |

Table 5.8. Performance results on learning algorithms using top 10 attributes.

## 5.4 Experiment 4: Multiple Classes

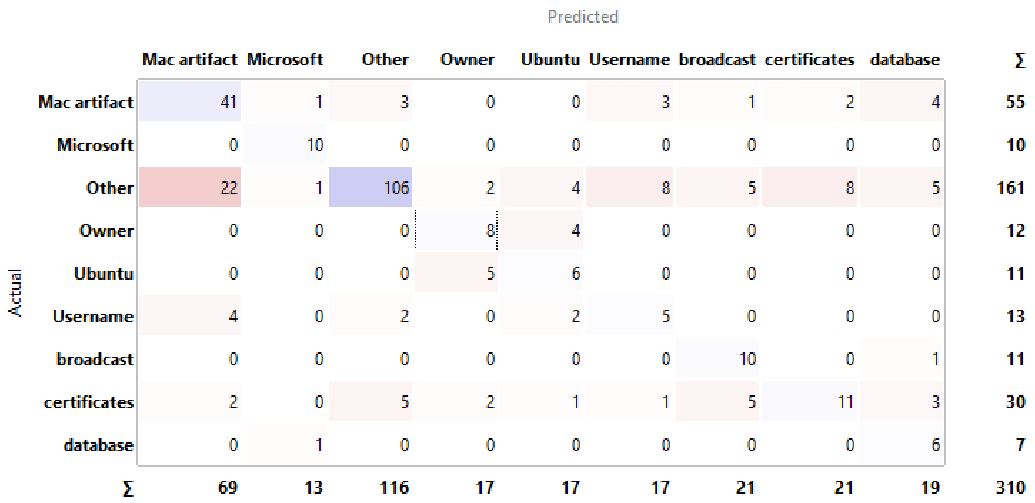
Table 5.9 shows the performance results for the learning algorithms when trained on the same dataset relabeled to use 9 classes. In comparing the results using 9 classes to our benchmark results in Table 5.2, it appears that there were not any major differences. SVM had the exact same scores across the board and Classification Tree saw some improvements in its AUC and F1 score. Naïve Bayes and Logistic Regressions decreased slightly in some measures and increased slightly in others. Based on this test alone, we could not definitively say that using multiple classes would work better.

| Method | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.818 | 0.958 | 0.552 | 0.471 | 0.667 |
| Classification Tree | 0.820 | 0.977 | 0.684 | 0.725 | 0.650 |
| Logistic Regression | 0.790 | 0.981 | 0.700 | 0.875 | 0.583 |
| SVM | 0.912 | 0.984 | 0.800 | 0.769 | 0.833 |

Table 5.9. Evaluation results for learning algorithms for 9 classes.

### 5.4.1 Merging Ubuntu and Owner Classes

To better understand our results, we looked at the classification decisions directly. Figure 5.3 shows the confusion matrix for Naïve Bayes for all 9 classes. The values in the diagonal from top-left to bottom-right represent the number of correct predictions. For the 'Mac artifact' class, out of the 55 actual 'Mac artifact' graphs, Naïve Bayes correctly predicted 41 as belonging to the 'Mac artifact' class. The 10 actual 'Microsoft' graphs were all correctly predicted as 'Microsoft' graphs. Out of the 161 'Other' graphs, 106 were predicted to be in the 'Other' class with 22 of the 'Other' graphs being wrongly predicted in the 'Mac artifact' class. 10 out of 11 actual 'broadcast' graphs were correctly predicted as well as 6 out of 7 'database' classes. The 'Username' and 'certificate' classes were misclassified the most often at less than half being predicted correctly. We found similar results with the other 3 learning algorithms.



Predicted

| Actual | Mac artifact | Microsoft | Other | Owner | Ubuntu | Username | broadcast | certificates | database | Σ |
|---|---|---|---|---|---|---|---|---|---|---|
| Mac artifact | 41 | 1 | 3 | 0 | 0 | 3 | 1 | 2 | 4 | 55 |
| Microsoft | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 |
| Other | 22 | 1 | 106 | 2 | 4 | 8 | 5 | 8 | 5 | 161 |
| Owner | 0 | 0 | 0 | 8 | 4 | 0 | 0 | 0 | 0 | 12 |
| Ubuntu | 0 | 0 | 0 | 5 | 6 | 0 | 0 | 0 | 0 | 11 |
| Username | 4 | 0 | 2 | 0 | 2 | 5 | 0 | 0 | 0 | 13 |
| broadcast | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 1 | 11 |
| certificates | 2 | 0 | 5 | 2 | 1 | 1 | 5 | 11 | 3 | 30 |
| database | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 7 |
| Σ | 69 | 13 | 116 | 17 | 17 | 17 | 21 | 21 | 19 | 310 |

Figure 5.3. Confusion matrix for Naïve Bayes using 9 classes.

Of particular interest in Figure 5.3 is the confusion it reveals between the ‘Owner’ and ‘Ubuntu’ classes. We saw that out of the 12 actual ‘Owner’ (Useful) graphs, 4 were wrongly predicted as belonging to the ‘Ubuntu’ class while 5 out of the 11 actual ‘Ubuntu’ graphs were wrongly classified as belonging to the ‘Owner’ class. This prompted us to look at the attribute values for each of the 9 classes. We found that the values for the ‘Ubuntu’ and ‘Owner’ classes were generally close to each other for the majority of the attributes. Figures 5.4 and 5.5 show the scatter plots with the different classes on the x-axis for each figure and the values for the attributes on the y-axis.

The transitivity (global clustering coefficient) values can be seen for the 9 different classes in Figure 5.4 and the size of a network’s maximal matching divided by the size of the network can be seen in Figure 5.5. Both figures show the ‘Owner’ (orange) and ‘Ubuntu’ (yellow) networks have attribute values in the same range. This proved to be the case with the majority of attributes. We hypothesize that the ‘Ubuntu’ networks may be similar to the ‘Owner’ networks in that they are both co-reference networks but leave this for future work.

Because of the similar topologies between the ‘Owner’ and ‘Ubuntu’ classes, we combined them into one class and tested our learning algorithms using the resultant 8 classes. Table 5.10 shows the results of the tests with ‘Ubuntu’ graphs classified as ‘Owner’ graphs.

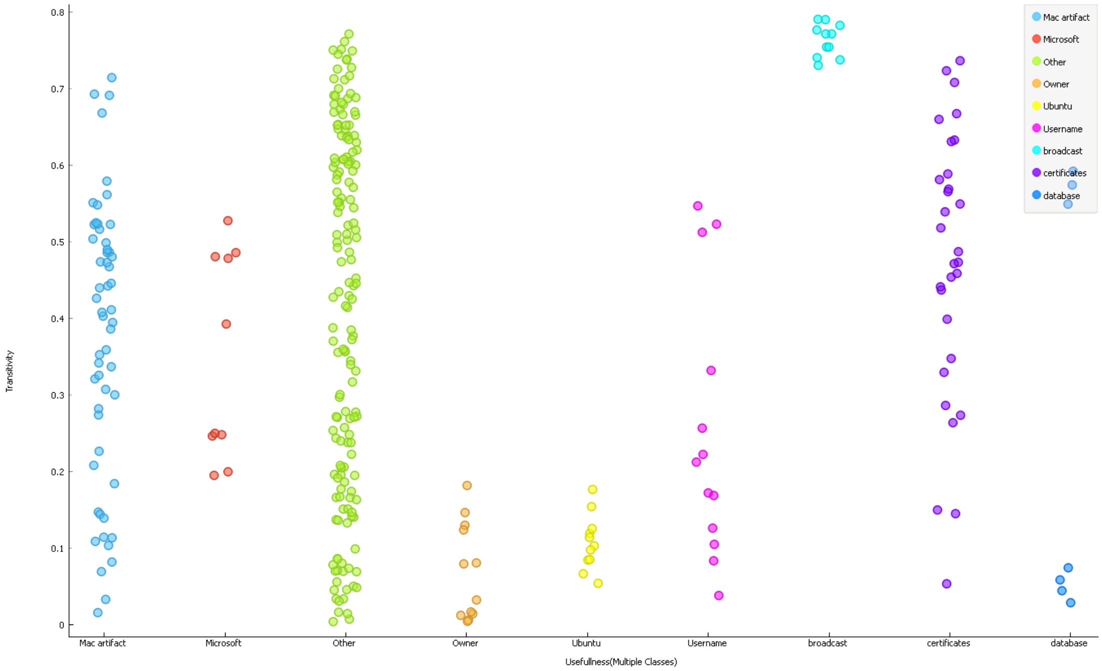


Figure 5.4. Scatter plot of Transitivity vs. 9 classes.

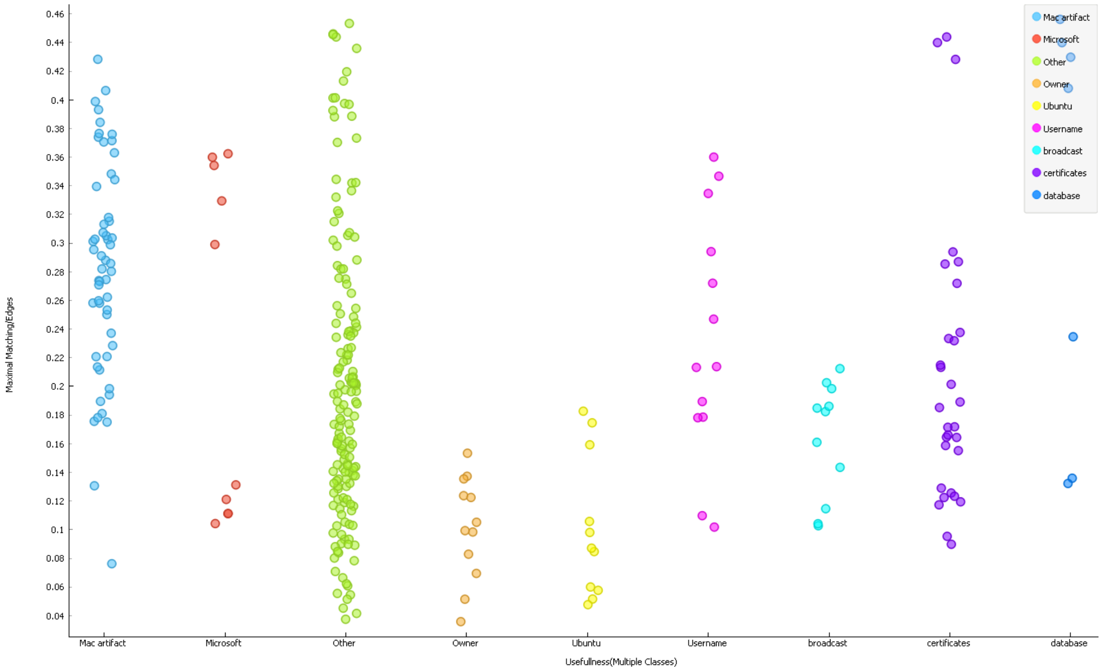


Figure 5.5. Scatter plot of Maximal Matching Size/Size of Network vs. 9 classes.

These were our best results yet. We saw increases for each algorithm for every score. Naïve Bayes saw an increase in its AUC to .986 and F1 score to .852. Classification Tree and Logistic Regression also performed well with their AUC increase to .925 and .933, respectively. SVM increased its AUC to .971 and F1 score to .898. We realize that combining the 'Ubuntu' networks with 'Owner' networks creates more networks that an analyst would have to filter through. However, a simple filter or stoplist could be created pre or post processing to simply leave out the 'Ubuntu' graphs.

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.986 | 0.974 | 0.852 | 0.742 | 1.000 |
| Classification Tree | 0.925 | 0.987 | 0.907 | 0.971 | 0.852 |
| Logistic Regression | 0.933 | 0.987 | 0.909 | 0.952 | 0.870 |
| SVM | 0.971 | 0.984 | 0.898 | 0.846 | 0.957 |

Table 5.10. Evaluation results for learning algorithms for 8 classes with 'Ubuntu' merged into 'Owner.'

Table 5.11 shows performance results of our final test, in which we return to a binary classification scheme but with 'Ubuntu' networks included under 'Useful.'

| **Method** | AUC | CA | F1 | Precision | Recall |
|---|---|---|---|---|---|
| Naïve Bayes | 0.980 | 0.965 | 0.797 | 0.663 | 1.000 |
| Classification Tree | 0.941 | 0.982 | 0.877 | 0.861 | 0.894 |
| Logistic Regression | 0.975 | 0.994 | 0.954 | 0.955 | 0.954 |
| SVM | 0.970 | 0.984 | 0.896 | 0.843 | 0.955 |

Table 5.11. Evaluation results for learning algorithms for 2 classes with 'Ubuntu' merged into 'Owner.'

The performance of all four learning algorithms roughly matches the scores attained using the 8 classes in our previous experiment. All four performed extremely well regardless of how many classes were used. If an analyst is interested in discovering more about what kind of software was being used, then the multiple class approach could be helpful.

# CHAPTER 6: Conclusions and Future Work

This final chapter presents the conclusions of our research. We provide answers to our questions from Chapter 4 and describe the contributions of our research. We conclude by presenting suggestions for future work.

## 6.1 Conclusions

The goal of our research was to find an effective and efficient way to assist digital forensic analysts in discovering email addresses that would be useful in determining the social network of the user(s) of the device from which they were extracted. Our research was unique in that it combined the fields of Digital Forensics, Network Science, Graph Theory, and Machine Learning in analyzing groups of email addresses as graphs created based on their physical location in storage on a device.

### 6.1.1 Experiment 1: Results

Our first experiment was conducted to answer our first question from Section 4.4:

***Is it possible to classify a network as being useful or not, based on the graph's underlying topological structure?***

Yes. We found that the four learning algorithms, Naïve Bayes, Classification Tree, Logistic Regression, and SVM were able to correctly classify a given network of email addresses with a fair amount of success. Naïve Bayes and SVM performed very well with AUCs at .905 and .868, respectively. Their scores were even higher when only networks with more than 20 nodes were tested. Out of the 4% of the total number of networks that were 'Useful,' Naïve Bayes and SVM captured 100% and 83.3% of them, respectively with SVM having a much better precision rate than Naïve Bayes at .769 as compared to .343. Naïve Bayes tended to over-classify networks as being 'Useful' when they were not. Classification Tree and Logistic Regression performed not as well, but they showed that they could still correctly identify over half of the 'Useful' networks.

### 6.1.2 Experiment 2: Results

Our second experiment was designed to answer the following:

***Does the size of the window used to create email graphs have an impact on our ability to classify them correctly?***
Both window sizes performed well. We found that some learning algorithms performed slightly better using the 128-byte window and that others performed slightly better when using a 256-byte window. In particular, we found that Logistic Regression performed exceptionally well with a 256-byte window with an AUC of .997 and a F1 score of .923. Logistic Regression had a slightly lower performance with a 128-byte window with an AUC of .913 and an F1 score of .833.

Although we weren't able to definitively declare that one window size outperformed the other, these results were encouraging as they showed that our underlying method was sound. The learning algorithms performed slightly better with both window sizes showing that slight perturbations did not have a negative influence on the results. Even more so, we saw a tighter clustering of attribute values for the better indicators when we tested the graphs with just one window size indicating that they would be even stronger indicators if we omitted the other window size.

### 6.1.3 Experiment 3: Results

We looked at the different attributes to determine which attributes gave the best indication that a graph would be useful or not. Our third experiment answered the following:

***What attributes are most effective for classifying the graphs in our dataset?***
Table 6.1 shows the 10 attributes (in no particular order) that were the best indicators of a network's usefulness along with the general value ranges for 'Useful' networks in the second column.

We found that when we used only these best attributes Naïve Bayes, Classification Tree, and SVM all performed slightly better than when run with all 41 collected attributes. Therefore, some of the 41 attributes were redundant, and some added noise. Also, using too many attributes can lead to over-fitting the results to the test data. If the data to be classified is different from the test data, the performance of the learning algorithm could change

| Attribute | Attribute Values of Useful Graphs |
|---|---|
| Density | <.05 |
| Average neighbor degree (r-normalized) | >.6 |
| Pearson coefficient | -.2 to .2 |
| Transitivity | <.2 |
| Highest Betweenness centrality | .5 to .8 |
| Maximal matching (divided by number of edges) | .05 to .15 |
| Maximal matching | >500 |
| Number of nodes (percentage from entire image) | .05 to .35 |
| Degree distribution (best fit) | Exponential |
| Degree Distribution value | >30 |

Table 6.1. 10 best attributes with their corresponding ranges for useful graphs.

dramatically. Logistic Regression, however performed poorly when being tested on fewer attributes.

We also tested the learning algorithms on 4 basic attributes that could be computed using the graph's size and order. These values could be pulled directly from the summary output produced when running the betographer program (see Section 2.4.2) as opposed to looking at each graph file. We found that Naïve Bayes had an AUC of .968 and a F1 score of .358. Naïve Bayes did have a recall rate of 1 again indicating that it if a graph was 'Useful', that it would classify that graph as being 'Useful'. It over-classified many, but could be valuable as an initial triage to discover potentially relevant networks. Classification Tree also performed well missing only 25% of the 'Useful' Networks but performing much better than Naïve Bayes in the number of predicted 'Useful' networks being actually 'Useful'.

### 6.1.4 Experiment 4: Results

We noticed that several graphs across our data set contained the same or similar structured email addresses and could be classified into their own classes. Our final experiment looked at using multiple classes and answered the following:

***Does our ability to correctly classify our graphs improve when we train against a multi-class labeling scheme, as opposed to a binary scheme in which the only labels are 'Useful' and 'Not-Useful.'?***

Yes. We initially tested the machine learning algorithms using 9 different classes and found that the learning algorithms performed roughly the same as when we tested the machine learning algorithms using 2 classes. We noticed that the 'Ubuntu' networks closely resembled the 'Owner' networks or 'Useful' graphs with the values of their attributes. The confusion matrix showed that the 2 classes were often misclassified against each other.

When we re-classified the 'Ubuntu' networks as 'Owner' and tested our learning algorithms again, we saw improvements for each learning algorithm for all performance measures. (note that the Ubuntu network can be easily screened out by an operator based on the domain name, but it provided extra testing data for us). Each algorithm had excellent AUCs of .986, .925, .933, and .971 for Naïve Bayes, Classification Tree, Logistic Regression, and SVM, respectively. All F1 scores were roughly .90 for each learning algorithm, which was an improvement of .1 to .4 depending on the algorithm. We also tested the learning algorithms with just 2 classes with the 'Ubuntu' graphs classified as being part of the 'Owner' or 'Useful' class and found similar results.

## 6.2 Contributions

We concluded that it was possible to effectively predict groups of email address as being useful or not-useful based solely on the topological features of the graphs created from the locations of those email addresses on a storage device and their proximity to each other. This technique could be applied to a variety of different storage devices, regardless of the operating system or email service. This could be a valuable toolset to an analyst conducting digital forensics in determining what email addresses are relevant and worth further examination in an investigation.

## 6.3 Future Work

Our research demonstrated great potential for discovering relevant email addresses on a storage device. There are, however, areas for further development. We recommend looking at a larger dataset, looking at other graph features, and looking at other Machine Learning methods.

### 6.3.1 Larger Dataset

Our dataset only consisted of 10 drives on which we conducted our experiments. Although our results looked promising, a larger data set would be required to definitively be able to declare that this technique would be useful on a more diverse set of drives. Furthermore, the majority of drives were from graduate students and faculty at an academic institution, more specifically in the computer science department. While this technique works well to discover social networks on computers of graduate level students/faculty, this does not generalize without testing our method on a more diverse data set.

We discovered that ‘Ubuntu’ graphs behaved similar to our ’User’ graphs. The reason is out of scope of this particular research but is an opportunity to further investigate how and why certain groups of email addresses appear on a device, and whether there is a legitimate reason to group them together with ‘Owner’ networks. Also, a larger sample could possibly find other types of networks that may be of use and could make up classes of their own. For example we saw a few graphs with email addresses that appeared to be from Facebook, Twitter, or from other online social network sites, but did not see enough of these types of graphs to attempt classify them.

Because our technique was operating system and device agnostic, useful graphs could be extracted from a large corpus in a automated manner saving time and resources. Those useful graphs could be combined into larger graphs and analyzed to see if there were larger networks of email addresses that formed.

### 6.3.2 Exploring Additional Features

We started with an initial intuition of what attributes would be best indicators. Our list grew as we used several algorithms that were supported by NetworkX. Network Science is an evolving field and there may be other attributes that we did not investigate that could be better indicators when classifying user networks. We looked at just one attribute at a time when determining whether it was a good or bad indicator. There are more advanced methods that look at multiple combinations of attributes and different calculations on the attributes. In addition, there are opportunities to optimize much of the feature extraction as many of the algorithms repeat some of the same procedures on the networks.

### 6.3.3 Testing Different Machine Learning Approaches

Unsupervised feature learning is another approach that looks at a network and returns the attributes that should be used to distinguish the graphs. These advanced techniques look at multiple combinations of attributes and different calculations on the attributes. Unsupervised machine learning using clustering could be tested on a larger collection of drives to determine possible classes. These classes could then be tested on an already labeled set. A multi-tier approach could be used as well where one removes what is definitely known to be not a useful graph, and then runs a machine learning algorithm on the remaining more balanced set of useful and not yet classified graphs. Our high recall results from Naïve Bayes suggest such a strategy could be effective here.

### 6.3.4 Conclusion

This method shows plenty of potential and there is much opportunity for it to be improved upon. We offered a new approach to discover relevant email addresses from a device. Our method takes a Network Science approach in looking at email addresses and their location in raw storage as nodes and edges on a graph. We showed that with a small collection of drives, that we could completely bypass the file system structure and determine which groups of email addresses that are co-located together could be relevant in an investigation.

# APPENDIX:

## A.1 Get Attributes.py

Below is the program to calculate attributes for a given graph.

```
import networkx as nx
import community

def highest_centrality(centralities):
    centralities=[(x,y) for (y,x) in centralities.items()]
    centralities.sort()
    centralities.reverse()
    return tuple(reversed(centralities[0]))

G = nx.read_gexf(file_name)

def find_average_neighbor_degree(graph_file):
    neighbor_degree_list = nx.average_neighbor_degree(G)
    sum_neighbor_degrees = sum(neighbor_degree_list())
    if nodes != 0:
        average_neighbor_degree = float(sum_neighbor_degrees)/nodes
    else:
        average_neighbor_degree = 0
    return (average_nd, min_nd, max_nd)

def highest_centrality(centralities):
    centralities=[(x,y) for (y,x) in centralities.items()]
    centralities.sort()
    centralities.reverse()
    return tuple(reversed(centralities[0]))
```

```
def find_max_min(some_list):
    some_list=[(x,y) for (y,x) in some_list.items()]
    some_list.sort()
    min_list = tuple(some_list[0])
    min_value = small_guy[0]
    some_list.reverse()
    max_list = tuple(reversed(some_list[0]))
    max_value = big_guy[1]
    return min_value, max_value

# Number of nodes
nodes = G.order()

# Number of edges
edges = G.number_of_edges(G)

# Average Degree
average_degree = 2 * nodes/edges

# Density
density = nx.density(G)

# Average Neighbor Degree
avg_neighbor_degree = find_average_neighbor_degree(G)

# Modularity
partition = community.best_partion(G)
modularity = community.modularity(partition, G)

# Average Clustering Coefficient
avg_clustering = nx.average_neighbor_degree(G)

# Pearson Coefficient
```

```
pearson = nx.degree_pearson_correlation_coefficient(G)

# Transitivity(Global Clustering Coefficient)
transitivity = nx.transitivity(G)

# Maximal Matching
maximal_matching_list = nx.maximal_matching(G)
max_matching = len(maximal_matching_list)

# Maximal Matching divided by number of edges
max_match_proportion = maximal_matching/edges

# Highest Degree Centrality
degree_centrality = nx.degree_centrality(G)
high_degree_centrality = highest_centrality(degree_centrality)

# Highest Betweenness Centrality
betweenness_centrality = nx.betweenness_centrality(G)
high_betweenness_centrality = highest_centrality(betweenness_centrality)

# Highest Closeness Centrality
closeness_centrality = nx.closeness_centrality(G)
high_closeness_centrality = highest_centrality(closeness_centrality)

# Maximimum Rich Coefficient
try:
    rich = nx.rich_club_coefficient(G)
except (nx.NetworkXAlgorithmError, nx.NetworkXError):
    rich = {0: 0}
results = find_max_min(rich)
max_rich = results[1]

# Mininmum Rich Coefficient
```

```
min_rich = results[0]

# Average Shortest Path Length
try:
    ASPL = nx.average_shortest_path_length(G)
except (nx.NetworkXAlgorithmError, nx.NetworkXError):
    ASPL = 0

# Maximum core
core_number = nx.core_number(G)
max_core = max(core_number.values())

# Number of elements in Max core
max_core_set = [v for v in core_number if core_number[v]==max_core]
max_core_size = len(max_core_set)

# Minimum core
min_core = min(core_number.values())

# Number of elements in Minimum core
min_core_set = [v for v in core_number if core_number[v]==min_core]
min_core_size = len(min_core_set)

# Diameter
ecc = nx.eccentricity(G)
diameter = max(ecc.values())

# Radius
radius = min(ecc.values())

# Center size
center = [v for v in ecc if ecc[v]==radius]
center_size = len(center)
```

```
# Periphery size
periphery = [v for v in ecc if ecc[v]==diameter]
periphery_size = len(periphery)

# Clique Number

clique_number = nx.graph_clique_number(G)
```

THIS PAGE INTENTIONALLY LEFT BLANK

# List of References

[1] M. Newman, *Networks: An Introduction*. Great Clarendon Street, Oxford: Oxford University Press, 2010.

[2] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, “Statistical properties of community structure in large social and information networks,” in *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, pp. 695–704.

[3] J. Green, “Constructing social networks from secondary storage with bulk analysis tools,” M.S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 2016.

[4] S. L. Garfinkel, “Digital media triage with bulk data analysis and bulk_extractor,” *Computers & Security*, vol. 32, pp. 56–72, 2013.

[5] J. R. Bradley and S. L. Garfinkel, “Bulk extractor user manual,” Naval Postgraduate School, Monterey, CA, Tech. Rep., mar 2013.

[6] A.-L. Barabási, *Network Science*. Cambridge, United Kingdom: Cambridge University Press, 2016.

[7] L. Euler, “Leonhard euler and the königsberg bridges,” *Scientific American*, vol. 189, no. 1, pp. 66–70, 1953.

[8] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*. Great Clarendon Street, Oxford: Oxford University Press, 1976.

[9] G. Chartrand and P. Zhang, *A first course in graph theory*. Mineola, New York: Courier Corporation, 2012.

[10] E. W. Weisstein. (2016). Unlabeled Graph. [Online]. Available: https://mathworld.wolfram.com/UnlabeledGraph.html

[11] E. W. Weisstein. (2016). Labeled Graph. [Online]. Available: https://mathworld.wolfram.com/LabeledGraph.html

[12] S. Fortunato, “Community detection in graphs,” *Physics Reports*, vol. 486, no. 3, pp. 75–174, 2010.

[13] N. Samay, S. Morrison, and A. Husseini, *Network Science Communities*. Cambridge, United Kingdom: Cambridge University Press, 2014.

[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, “Fast unfolding of communities in large networks,” *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

[15] T. Opsahl, “Triadic closure in two-mode networks: Redefining the global and local clustering coefficients,” *Social Networks*, vol. 35, no. 2, pp. 159–167, 2013.

[16] J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertesz, “Generalizations of the clustering coefficient to weighted complex networks,” *Physical Review E*, vol. 75, no. 2, p. 027105, 2007.

[17] J. G. Foster, D. V. Foster, P. Grassberger, and M. Paczuski, “Edge direction and the structure of networks,” *Proceedings of the National Academy of Sciences*, vol. 107, no. 24, pp. 10 815–10 820, 2010.

[18] M. E. Newman, “Mixing patterns in networks,” *Physical Review E*, vol. 67, no. 2, p. 026126, 2003.

[19] L. Wang, T. Lou, J. Tang, and J. E. Hopcroft, “Detecting community kernels in large social networks,” in *2011 IEEE 11th International Conference on Data Mining*. IEEE, 2011, pp. 784–793.

[20] S. Vishwanathan, K. M. Borgwardt, N. N. Schraudolph *et al.*, “Fast computation of graph kernels,” in *NIPS*, 2006, vol. 19, pp. 131–138.

[21] T. Gärtner, “A survey of kernels for structured data,” *ACM SIGKDD Explorations Newsletter*, vol. 5, no. 1, pp. 49–58, 2003.

[22] U. Brandes, “A faster algorithm for betweenness centrality*,” *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[23] L. C. Freeman, “Centrality in social networks conceptual clarification,” *Social networks*, vol. 1, no. 3, pp. 215–239, 1978.

[24] M. E. Newman, “The mathematics of networks,” *The New Palgrave Encyclopedia of Economics*, vol. 2, no. 2008, pp. 1–12, 2008.

[25] G. Li, M. Semerci, B. Yener, and M. J. Zaki, “Effective graph classification based on topological and label attributes,” *Statistical Analysis and Data Mining*, vol. 5, no. 4, pp. 265–283, 2012.

[26] E. Alpaydin, *Introduction to machine learning*. Cambridge, Massachusetts: MIT press, 2014.

[27] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, 2003, vol. 2.

[28] I. The MathWorks. (2016). Unsupervised Learning. [Online]. Available: http://www.mathworks.com/discovery/unsupervised-learning.html

[29] T. Mitchell, "Generative and discriminative classifiers: naive bayes and logistic regression, 2005," *Manuscript available at http://www. cs. cm. edu/~ tom/NewChapters. html*, 2016.

[30] A. P. Dawid, "Conditional independence in statistical theory," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–31, 1979.

[31] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[32] T. G. Tape. The Area Under an ROC Curve. [Online]. Available: http://gim.unmc.edu/dxtests/roc3.htm

[33] M. Bastian, S. Heymann, M. Jacomy *et al.*, "Gephi: an open source software for exploring and manipulating networks." *ICWSM*, vol. 8, pp. 361–362, 2009.

[34] D. A. Schult and P. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conferences (SciPy 2008)*, 2008, vol. 2008, pp. 11–16.

[35] D. Conway and A. Hagberg. (2011). Module II - Why do SNA in NetworkX. [Online]. Available: https://github.com/drewconway/NetworkX_Intro_Materials/blob/master/2-Why_Do_SNA_with_NX/module_II.pdf

[36] J. Demšar and B. Zupan, "Orange: Data mining fruitful and fun-a historical perspective," *Informatica*, vol. 37, no. 1, 2013.

[37] J. Demšar, T. Curk, A. Erjavec, Č. Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič *et al.*, "Orange: data mining toolbox in python," *Journal of Machine Learning Research*, vol. 14, no. 1, pp. 2349–2353, 2013.

[38] O. D. Mining. (2015). Orange Visual Programming. [Online]. Available: http://docs.orange.biolab.si/3/visual-programming/index.html

[39] N. Willis. (2012). Data mining with Orange. [Online]. Available: https://lwn.net/Articles/504741/

[40] E. Z. Goodnight. (2014, Sep.). Email: What the difference in POP3, IMAP, and Exchange. [Online]. Available: http://www.howtogeek.com/99423/

[41] S. L. Garfinkel, "Digital forensics research: The next 10 years," *Digital Investigation*, vol. 7, pp. S64–S73, 2010.

[42] M. Gielen, “Prioritizing computer forensics using triage techniques,” 2014.

[43] V. K. Devendran, H. Shahriar, and V. Clincy, “A comparative study of email forensic tools,” *Journal of Information Security*, vol. 6, no. 2, p. 111, 2015.

[44] N. C. Rowe, “Identifying forensically uninteresting files using a large corpus,” in *International Conference on Digital Forensics and Cyber Crime*. New York City, NY: Springer, 2013, pp. 86–101.

[45] L. G. Jeub, P. Balachandran, M. A. Porter, P. J. Mucha, and M. W. Mahoney, “Think locally, act locally: Detection of small, medium-sized, and large communities in large networks,” *Physical Review E*, vol. 91, no. 1, p. 012821, 2015.

[46] M. E. Newman and M. Girvan, “Finding and evaluating community structure in networks,” *Physical Review E*, vol. 69, no. 2, p. 026113, 2004.

[47] W. W. Zachary, “An information flow model for conflict and fission in small groups,” *Journal of Anthropological Research*, pp. 452–473, 1977.

[48] M. E. Newman, “Fast algorithm for detecting community structure in networks,” *Physical Review E*, vol. 69, no. 6, p. 066133, 2004.

[49] J. Huang and C. X. Ling, “Using auc and accuracy in evaluating learning algorithms,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 3, pp. 299–310, 2005.

# Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California